Fault Tolerant Real Time Dynamic Scheduling Algorithm For Heterogeneous Distributed System

Aser Avinash Ekka

Department of Computer Science & Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Orissa, INDIA

2007

Fault Tolerant Real Time Dynamic Scheduling Algorithm For Heterogeneous Distributed System

Thesis submitted in partial fulfillment
of the requirements for the degree of

MASTER OF TECHNOLOGY (RESEARCH)

by

Aser Avinash Ekka

Thesis Advisor: Bibhudatta Sahoo

National Institute of Technology Rourkela
Department of Computer Science & Engineering
Rourkela-769 008, Orissa, INDIA

To my family

National Institute of Technology

Rourkela

Certificate

This is to certify that the work in the thesis entitled **"Fault Tolerant Real Time Dynamic Scheduling Algorithm For Heterogeneous Distributed System "** submitted by **Mr. Aser Avinash Ekka** is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology (Research) in Computer Science and Engineering during the session 2005-2007 in the department of Computer Science and Engineering, National Institute of Technology Rourkela. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Date:
Place: NIT Rourkela

Bibhudatta Sahoo
Senior Lecturer
Department of Computer Science & Engg.
National Institute of Technology
Rourkela-769008

Acknowledgements

The person I am most indebted to is Bibhudatta Sahoo, Senior Lecturer, CSE department of NIT Rourkela. I think when he agreed to be my advisor, he had no idea what he was in for. He has been most patient with me, setting aside precious time to coach me, even beyond machine learning. He has even helped me learn a new "model" of what research is and isn't, and re-evaluate my outlook. He is at once mentor, colleague, and friend, allowing me to freely embrace myself. For all these, and more, thank you, Bibhu Sir.

Prof. (Dr.) S.K. Jena, offered timely and supportive counsel and has always been eminently approachable. Along with Prof. (Dr.) B. Majhi and Asstt. Prof. (Dr.) A.K. Turuk supervised my tutoring at CSE department of NIT Rourkela. My sincere thanks goes to Prof. S. K. Rath and Asstt. Prof. (Dr.) D.P. Mohapatra for motivating me to work harder. This work has helped me to grow both professionally and personally.

I would like to express my thanks to the other postgraduate students at NIT Rourkela, notably Pankaj Kumar Sa, Dillip Puthal, Sunil Pattnaik, Manas Ranjan Meher, Saroj Mishra, Ranghunath and Sushant Pattnaik for their peer support. Indeed, given the atmosphere of congeniality I have enjoyed during my research, I must offer a blanket thank you to all the staff particularly Mr. Manoj Pattnaik, Technical Asstt. and students with whom I have worked at NIT Rourkela.

And, finally, I must thank my family, have provided me with the love, stability, and practicality that has allowed me to persist with studying. Such a connection is valuable in itself.

Funding for this research was provided by CS-HDS Project Grant 2005-2008, Ministry of HRD, Govt. of India.

Aser Avinash Ekka
Roll No. 60506002

Fault Tolerant Real Time Dynamic Scheduling Algorithm for Heterogeneous Distributed System

Aser Avinash Ekka
NIT Rourkela

Abstract

Fault-tolerance becomes an important key to establish dependability in Real Time Distributed Systems (RTDS). In fault-tolerant Real Time Distributed systems, detection of fault and its recovery should be executed in timely manner so that in spite of fault occurrences the intended output of real-time computations always take place on time. Hardware and software redundancy are well-known effective methods for fault-tolerance, where extra hard ware (e.g., processors, communication links) and software (e.g., tasks, messages) are added into the system to deal with faults. Performances of RTDS are mostly guided by efficiency of scheduling algorithm and schedulability analysis are performed on the system to ensure the timing constrains.

This thesis examines the scenarios where a real time system requires very little redundant hardware resources to tolerate failures in heterogeneous real time distributed systems with point-to-point communication links. Fault tolerance can be achieved by scheduling additional ghost copies in addition to the primary copy of the task. The method proposed here extends the traditional distributed recovery block (DRB) based fault tolerant scheduling approach for real time tasks. Our algorithms are mostly based on software redundancy, and ensures that the parallel updation of backup task works better in case of transient overload and handles permanent, transient and timing fault.

The performance of our proposed schemes has been verified using most common quality metrics, guarantee ratio and throughput. An attempt has been made to propose a scheduling scheme based upon fixed length coding using DNA algorithm. Which can be used to design an efficient Fault Tolerant algorithm for real time distributed system using DRB. The research has resulted in development of a Fault Tolerant Real Time Dynamic Scheduling Algorithm For Heterogeneous Distributed which also leads into building an Adaptive Fault Tolerant Dynamic Scheduling Algorithm.

The development in the thesis is genuinely supported by detailed literature survey and mathematics preliminaries leading to the proposed model of fault tolerant algorithm. For shake of continuity each chapter has its relevant introduction and theory. The work is also supported by list of necessary references. Attempt is made to make the thesis self-content.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Real-time computer systems are required by their environments to produce correct results not only in their values but also in the times at which the results are produced better known as timeliness. In such systems, the execution times of the real-time programs are usually constrained by predefined time-bounds that together satisfy the timeliness requirements. To obtain parallel processing performance and to increase reliability, distributed architectures having processors interconnected by a communications network are increasingly being used for real-time systems giving rise to real time distributed systems (RTDS). In such real-time distributed systems, programs assigned to different processors interact by sending and receiving messages via communication channels

A failure in a real-time system can result in severe consequences. Such critical systems should be designed a priori to satisfy their timeliness and reliability requirements. To guarantee the timeliness, one can estimate the worst-case execution times for the real-time programs and check whether they are schedulable, assuming correct execution times. Despite such design, failures can still occur at run-time due to unanticipated system or environment behaviors.

Fault-tolerant computing deals with building computing systems that continue to operate satisfactorily in the presence of faults. A fault-tolerant system may be able to tolerate one or more fault-types including - (i) transient, intermittent or permanent hardware faults, (ii) software and hardware design errors, (iii) operator errors, or (iv) externally induced upsets or physical damage. An extensive methodology has been developed in this field over the past thirty years, and a number of fault-tolerant machines have been developed - most of them dealing with random hardware faults, while a smaller number deal with software, design and operator faults to varying degrees. A large amount of supporting research has been reported. Fault tolerance and dependable systems research covers a wide spectrum of applications ranging across

embedded real-time systems, commercial transaction systems,transportation systems, and military/space systems - to name a few. The supporting research includes system architecture, design techniques, coding theory, testing,validation, proof of correctness, modeling, software reliability, operating systems, parallel processing, and real-time processing. These areas often involve widely diverse core expertise ranging from formal logic, mathematics of stochastic modeling, graph theory,hardware design and software engineering.

Redundancy has long been used in fault-tolerant and adaptive systems. However, redundancy does not inherently make a system fault-tolerant and adaptive; it is necessary to employ fault-tolerant methods by which the system can tolerate hardware component failures, avoid or predict timing failures, and be reconfigured with little or graceful degradation in terms of reliability and functionality.

Early error detection is clearly important for real-time systems; error is an abbreviation for erroneous system state, the observable result of a failure. The error detection latency of a system is the interval of time from the instant at which the system enters an erroneous state to the instant at which that state is detected. Keeping the error detection latency small provides a better chance to recover from component failures and timing errors, and to exhibit graceful reconfiguration. However, a small latency alone is not sufficient; fault-tolerant methods need to be provided with sufficient information about the computation underway in order to take appropriate action when an error is detected. Such information can be obtained during system design and implementation. In current practice, the design and implementation for real-time systems often does not sufficiently address fault tolerance and adaptiveness issues.

The performance of a RTDS can be improved by proper task allocation and an effective uniprocessor scheduling. In this thesis a brief study of the existing task allocation and uniprocessor scheduling schemes has been presented and work has also been done to find an appropriate schemes for allocation and scheduling in distributed system.

1.2 Dependability in RTDS: A case study

Tradeoffs among system performance and with respect to reliability are becoming increasingly important. Hence, reliability measurement in Real Time Distributed System is of an important use. The list of reliable Real Time Distributed Systems is very vast. We have therefore restricted our list to only a handful of applications where incapability to deal with faulty components leads to hazardous results.

Computer On-Board an Aircraft: In many modern aircraft the pilot can select an “auto pilot”option. As soon as the pilot switches to this mode, an on-board computer takes over all control of the aircraft including navigation, take-off , and landing of the aircraft. In the “auto pilot”, the computer periodically samples velocity and acceleration of the

aircraft. From the sampled data, the on-board computer computes X,Y, and Z co-ordinates of the current aircraft position and compare them with pre-specified track data. It computes the deviation from the specified track values and take any corrective actions that may be necessary. In this case, the sampling of various parameters, and their processing need to be completed within a few microseconds.

Missile Guidance System: A guided missile is one that is capable of sensing the target and homes onto it. In a missile guidance system, missile guidance is achieved by a computers mounted on the missile. The mounted computer computes the deviation from the required trajectory and effects track changes of the missile to guide it onto the target. The time constraint on the computer based guidance system is that sensing and the track correction tasks must be activated frequently enough to keep the missile from straying from the target. Tasks are typically required to be completed within few hundreds of microsecond or even lesser time. If the computer on the missile goes faulty then means should integrated by which this faulty systems can be compensated, which is clearly possible by means of good fault tolerant technique.

Internet and Multimedia Applications: Important uses of real-time systems in multimedia and Internet applications include: video conferencing and multimedia multicast, Internet router and switches. In a video conferencing application, video and audio signals are generated by cameras and microphones, respectively. The data are sampled at certain prespecified frame rate. These are then compressed and sent as packets to the receiver over a network. At the receiver-end, packets are ordered, decompressed and then played. The time constraint at the receiver end is that the receiver must process and play the received frames at predetermined constant rate.

Steps to achieve dependability and its measure can be of great value in engineering Real Time Distributed Systems.

1.3 Literature Review

In many application domains there is a strong dependency on Real TIme Distributed Computing Systems and the availability of a continuous service is often extremely important. Most of these systems are designed to be fault-tolerant. Philip H. Enslow Jr. [45] (1977) outlines four system characteristics serve as motivations for the continued development of parallel processing systems (1) Throughout (2) Flexibility (3) Availability and (4) Reliability.

The importance of Real-Time Distributed Computing Fault tolerant Building-Blocks in Realization of Ubiquitous Computing has been discussed by K.H. Kim [51]. Kim further outlines issues and advances in his paper [47, 48, 49]. Arora And Gouda (1993) [7] formally define tolerating a class of fault in terms of “Closure ”and “Convergence ”. Lala and Harper [55] (1994) describe aspects of safety critical in real time applications,

the architectural principles and techniques to address these unique requirements are described. Paper by Felix C. Gartner [33] (1999) use a formal approach to define important terms like fault, fault tolerance, and redundancy. This leads to four distinct forms of fault tolerance and to two main phases in achieving them: detection and correction. Avizienis, Laprie and Randell [8] give the main definitions relating to dependability, a generic concept including as special case such attributes as reliability, availability, safety, confidentiality, integrity, maintainability, etc. Heimerdinger and Weinstock (1992) in their technical report provide a conceptual framework for system fault tolerance with examples [37].

Redundancy is the heart of fault tolerance where multiple copies of a task is executed concurrently at different site if one of the version fails the rest take its place. Ghosh, Melhem and Daniel [94, 30] propose a fault tolerance scheduling algorithm in distributed systems which schedules several backup tasks overlapping one another and dynamically deallocates the backups as soon as the original tasks complete executions, thus increasing the utilization of processors. Gupta, Manimaran and Murthy claim that in general, the Primary-Backup overlapping approach performs better than both primary-backup exclusive and primary-backup concurrent [34] (1999). A successful fault tolerant algorithm with the EDF algorithm for multiprocessors running in parallel and executing real-time applications based on a popular fault-tolerant technique called primary/backup (PB) fault tolerance has been described in [68] (2006). A solution corresponding to a software implemented fault-tolerance, by mean of software redundancy of algorithm´s operations and timing redundancy of communications is described by Girault, Lavarenne, Sighireanu and Sorel [31] (2001).

A dual-cluster distributed real-time system model with four states to improve the reliability of distributed real-time systems has been described by Zhiying Wang, Jeffrey J. P. Tsai and Chunyuan Zhang [117] (2000). The system employs cluster switch, output arbitration and fault detection to increase its reliability. C. Tanzer and S. Poledna and E. Dilger and T. Fhrer in [105] describe the conceptual model for, and the implementation of, a software fault-tolerance layer (FT-layer) for distributed fault-tolerant hard real time systems. This FT-layer provides error detection capabilities, fault-tolerance mechanisms based on active replication, and the interface between the application software running on a node of the distributed system and the communication services. Considering reliability cost as the main performance metric Qin, Han, Pang, Li and Jin (2000) [82] presented two fault-tolerant scheduling algorithms for periodic real-time tasks in heterogeneous distributed systems. Problem of scheduling a periodic real-time system on identical multiprocessor platforms, where the tasks considered may fail with a given probability has been dealt by Berten, Goossens and Jeannot (2006) [11]. Park and Yeom (2000) [74] presents a new checkpointing coordination scheme to reduce the overhead of checkpointing coordination. Wong and Dillon (1999) [112] propose a fault tolerant and high performance model for Internet. Jaime Cohen (2001) presents a graph partitioning which can be applied to many application related to fault

tolerant computing.

The execution environment of most of the real time distributed systems is imperfect and the interaction with the external world introduces additional unpredictability; design assumptions can be violated at run time due to unexpected conditions such as transient overload, application of formal techniques or scheduling algorithms in turn requires assumptions about the underlying system and it may be infeasible to verify formally some properties; thus necessitating run time checks. Chodrow, Jahanian and Donner [18] (1991) describe a model of a run time environment for specifying and monitoring properties of a real time system. In [96] Beth A. Schroeder outlines the concepts and identifies the activities that comprise event based monitoring, describing several representative monitoring systems. Results show that a fast failure detector service (implemented using specialized hardware or expedited message delivery) can be an important tool in the design of real-time mission-critical systems [4] (2002). S. Oh and G. MacEwen (1992) [73] have introduced a monitoring approach based on fail-signal processors, which allows continuous observation of the functional and timing behaviour of application processors. Haban and Shin (1990) [35] propose to use a real-time monitor as a scheduling aid. Different methods by which fault tolerance can be provided in a system has been discussed by Anderson [6]. Jalote presents an excellent framework for fault tolerance in distributed systems [42]. Cristian provided a survey of the issues involved in providing fault-tolerant distributed systems [22]. Litke, Skoutas, Tserpes and Varvarigou present an efficient scheme based on task replication, which utilizes the Weibull reliability function for the Grid resources so as to estimate the number of replicas that are going to be scheduled in order to guarantee a specific fault tolerance level for the Grid environment (2006) [60]. Performance of real time distributed system can be improved from scheduling of task aspect.

The scheduling problem in real time systems has been explained in [15, 90, 32, 53, 102, 98]. A heuristic dynamic scheduling scheme for parallel real-time jobs, modeled by directed acyclic graphs (DAG) where parallel real time jobs arrive at a heterogeneous system following a Poisson process, in a heterogeneous system is presented in [81]. Assignment and Scheduling Communicating Periodic Tasks in Distributed Real-Time Systems has been dealt by Peng, Shin and Abdelzaher (1997) [76, 39]. Ramamritham (1995) [87] also deals with allocation and scheduling of precedence-related periodic tasks. A sufficient condition for the schedulability of preemptable, periodic, hard-real-time task sets using the very simple static-priority weight-monotonic scheduling scheme has been presented by Ramamurthy and Moir (2001) [88].

1.4 Motivation

A “fault-tolerant system” is one that continues to perform at desired level of service in spite of failures in some components that constitute the system.

Fault tolerance has always been around from NASA´s deep space probe to Medical Computing devices (e.g. pacemaker). But this had been overlooked until now. With the real world problem more inclined and dependent on computer the need fault tolerance has also increased. Extreme fault tolerance is required in car controllers (e.g. anti-lock brakes). Commercial servers (databases, web servers), file servers, etc. require high fault tolerance. Application running on even Desktops, laptops (really!), PDAs, etc. also require some fault tolerance. Fault tolerance is needed in systems such as telephone systems, Banking systems, e.g. ATM and Stock market, where reliability is essential. Manned and unmanned space borne systems, Aircraft control systems, Nuclear reactor control systems and Life support systems which are critical and life critical systems definitely need fault tolerance. Hence, this thesis takes up the work of building a fault tolerant system for periodic tasks in real time distributed platform.

1.5 Problem Statement

Fault tolerance is a fundamental aspect of building dependable systems, and can consume a major part of building a system. This is particularly true for distributed system working in a real time environment, which have to be rigorously performing against all odds to ensure their correctness. Further, real time distributed systems must not only be tested for functional correctness, but also for timeliness.

To achieve the desired level of dependability in application correctness, full ability to tolerate faults is strongly desirable for real-time systems. This means that all anticipated behaviors must be dealt with. In general, full fault coverage is not feasible, because of the large amount of test cases that must be designed, executed, and analyzed. Therefore, to enable testing with full fault coverage, some constraints on application behavior must be introduced. The goal of this thesis to identify techniques that can, according to a specific criteria, providing tolerance to a selected kind of faults specifically, permanent, timing and transient faults for event triggered real time distributed system.

The processors are assumed to fail in the fail-stop manner and the failure of a processor can be detected by other processor. All periodic tasks arrive at the system having different period and are ready to execute any time within each period. We further assume that all periodic tasks have deadlines greater than its period and execution time and their deadlines have to be met even in the presence of processor failures. We define a tasks meeting its deadline as either its primary copy or its backup copy finishes before or at the deadline. Because the failure of processors is unpredictable and there is no optimal dynamic scheduling algorithm for real time distributed system scheduling, we focus on dynamic scheduling algorithms to ensure that the deadlines of tasks are met even if some of the processors might fail. The fault tolerant scheduling problem is defined as follows:

Fault Tolerant Scheduling Problem: A set of n periodic tasks $\xi = \{T_1, T_2, \ldots T_n\}$ is to be scheduled on a number of processors, task i will be represented as T_i until and unless specified. For each task T_i, there are a primary copy Pri_{T_i} and a backup copy $Back_{T_i}$ associated with it. The computation time of a primary copy Pri_{T_i} is denoted as c_i, which is the same as the computation time of its backup copy $Back_{T_i}$. The tasks may be independent or dependent of each other. The fault tolerant scheduling requirements are given as follows:

1. Each task is executed by one processor at a time and each processor executes one task at a time.

2. All periodic tasks should meet their deadlines.

3. Maximize the number of processor failures to be tolerated.

4. For each task T_i, the primary task Pri_{T_i} or the backup $Back_{T_i}$ is assigned to only one processor for the duration of c_i and can be preempted once it starts, if their is a task with early deadline than the presently executed task.

The processors are assumed to be not identical i.e. they are heterogeneous. Requirement (1) specifies that there is no parallelism within a task and within a processor. Requirement (2) dictates that the deadlines of periodic tasks should be met, maybe at the expense of more processors. Requirement (3) is a very strong requirement. The primary and backup tasks should be scheduled on different processors such that any one or more processor failure will not result in the missing of the deadlines of the periodic tasks. Requirement (4) implies that tasks are preemptive. A processor is informed the failure of other processors during the execution of a task. Also, care has to be taken to ensure that exactly one of the two copies of a task is executed to minimize the wasted work.

Since no efficient scheduling algorithm exists for the optimal solution of the fault-tolerant real-time multiprocessor scheduling problem as defined above, we resolve to a heuristic approach.

1.6 Thesis Outline

In this we have worked upon developing a fault tolerant dynamic scheduling algorithm for distributed systems working in an real time environment. The work leads upto developing an adaptive fault tolerant scheme for the distributed system. The thesis has been divided into six chapters and has been arranged as follows. **Chapter 1** gives a brief description of fault tolerance in Real Time Distributed System and related works by various researchers in the said area. **Chapter 2** describes the architecture of Real Time Distributed System and periodic task model and its characteristics . We have summarized the different type of faults, taxonomy of faults and different fault

tolerant techniques. Scheduling algorithms in RTDS has been briefly discussed in **Chapter 3**. An appropriate assignment scheme has been evaluated in this chapter as well as the multivariate framework for distributed systems has been also proposed. **Chapter 4** contains the evaluation of proposed DRB based fault tolerant algorithms in RTDS. We have described three proposed algorithms that provides better fault-tolerance capability for periodic task on heterogeneous distributed system. **Chapter 5** gives the evolutionary concepts for task assignment. **Chapter 6** finally concludes the thesis. Future work has been mentioned in **Chapter 7**.

Chapter 2

Real Time Distributed System: Model, Performance Metric and Faults

2.1 Introduction

The focus of this thesis is on providing dependability by means of fault tolerance in distributed systems, operating in real time environment. We give the model of the real time distributed system that is used for the entire thesis. The goal of fault tolerance in distributed systems is often to ensure that some property, or service, in the logical model is preserved despite the failure of some components in the physical system.

A system is called real-time systems, when a quantitative expression of time to describe the behaviour of the system. There are a large variety of real time systems but all have common characteristics which differentiate them from non-real time systems.

1. Time Constraint: One very common form of time constraint is deadline associated with tasks. A task deadline specifies the time before which the task must complete and produce results. It is the responsibility of the RTOS, schedulers particularly, to ensure that all tasks meet their respective deadline.

2. Safety-Criticality: For traditional non-real time systems safety and reliability are independent issues. However, in many real time systems these two issues are intricately bound together making them safety-critical.

The key properties of distributed systems are the scalability and autonomous nature of various nodes. Distributed systems are different from parallel systems, where nodes are closely coupled, i.e. they are not autonomous. Unlike parallel systems distributed systems are loosely coupled and do not have global clock driving all the nodes. Major atomic components of the distributed systems are processor, communication network, clocks, software, and non-volatile storage. Fault tolerance scheme aims to mask faults in these components.

In distributed systems maximizing utilization of resources is a major concern hence task are assigned/mapped and migrated among participating nodes so that overall performance and utilization of the system is increased. Hence apart from the local schedulers at the nodes such system, which deal with real time tasks, need a feasible assignment scheme so that the tasks are executed on or within their deadline.

An abstract model of the RTDS has to be created in order to formalize the system behavior of a real time distributed system within time domain. The abstract model describes (1) the system architecture (2) the workload model, which describes the application running on that architecture, plus the timing constraint the application must fulfill.

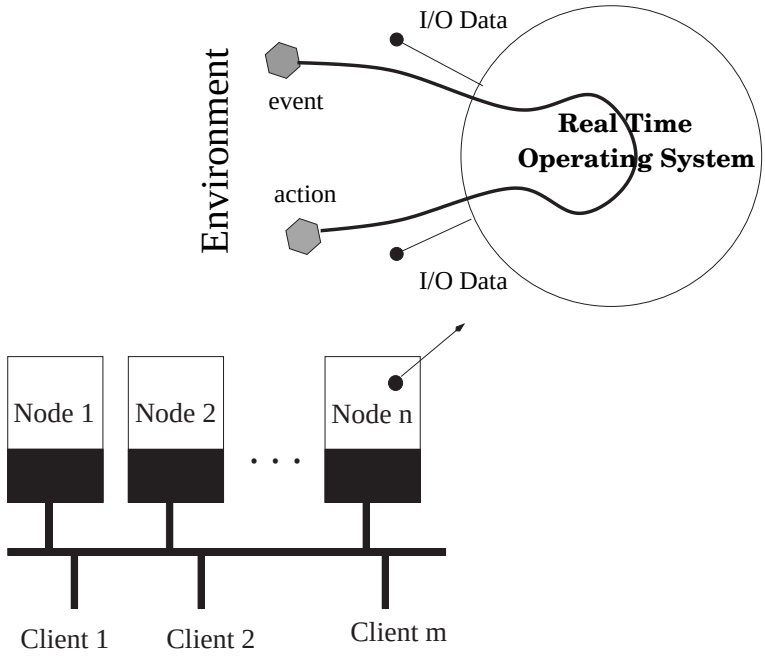


Figure 2.1: Real Time Distributed System Architecture

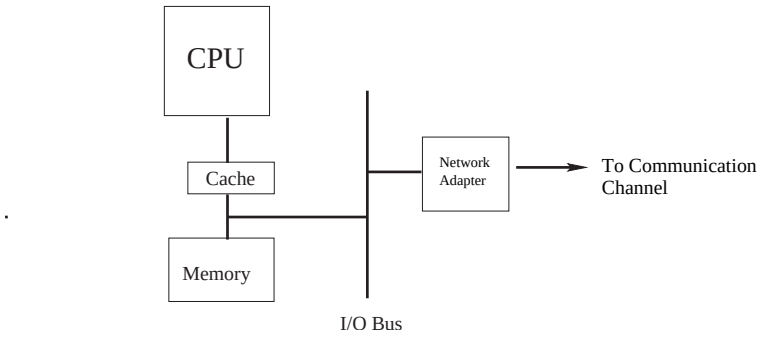


Figure 2.2: Node Architecture

2.2 System Model

In our study we define a real time distributed system as follows;

Definition 1 (Real Time Distributed System). *A real time distributed system consists of computation nodes connected by a real time network where the messages transfer time between the nodes is bounded. Each node may execute a distributed real time kernel which provides the local and remote interprocess communication and synchronization as well as the usual process management functions and input/output management.*

A Real Time Distributed System consists of a number of processors interconnected by a network as shown in Figure 2.1. The configuration of each node is further shown in Figure 2.2. This architecture is described by a tuple $\Pi = (P, \kappa)$, where $P = \{P_1, P_2, .., P_n\}$ a set of heterogeneous computing systems (*HCS*) and κ defines the type of communication channel. Channels from a node P to node Q represents the fact that P can send messages to Q. Channel are assumed to have infinite buffer and are assumed to be error-free. Further, we assume that channels deliver messages in the order in which they are sent. Each computing system is defined by a tuple $P_i = (\mu^{P_i}, \nu)$ where μ^{P_i} is the execution rate of P_i measured in MFLOPs and ν is the memory attached to P_i in this thesis the memory size is considered in terms of queue length of respective processors. Menasce and Almeida [67] proposed two distinct forms of heterogeneity in high-performance computing systems. Function-level heterogeneity exists in systems that combine general-purpose processors with special-purpose processors, such as vector units, floating-point co-processors, and input/output processors. With this type of heterogeneity, not all of the tasks can be executed on all of the function units. Performance-level heterogeneity, on the other hand, exists in systems that contain several interconnected general-purpose processors of varying speeds. In these systems, a task can execute on any of the processors, but it will execute faster on some than on others. In this work we have considered only performance level heterogeneous computing systems (HCS).

The network in our model provides full connectivity through either a physical link or a virtual link. A HCS communicates with other HCS through message passing, and the communication time between two tasks assigned to the same HCS is assumed to be zero. The network κ is an arrangement of nodes by means of communication channel. $\kappa = (ch_1, ch_2, .., ch_m)$ where ch_i is a communication channel. Network performance can be measured in two fundamental ways: *bandwidth* and *latency*.

$$Transmit = Size/Bandwidth \tag{2.1}$$

$$Propagation = Distance/Speed\ of\ Light \tag{2.2}$$

$$Latency = Propagation + Transmit + Queue \tag{2.3}$$

2.3 Workload Model

All kinds task and process management strategies in any system including a real time distributed system the most important term to define is the real-time task. For different domains of computer science the exact meaning varies greatly. Terms such as application, task, sub task, task force and agent are used to denote the same object in some instances, and yet, have totally different meanings in others. In order to be consistent in our analysis we have chosen the following basic definition of a real-time task and a timing diagram of which has been shown in Figure 2.3.

Definition 2 (Real Time Task). *The basic object of scheduling is an instance of a task. Usually the fundamental properties are arrival time, period over which the task is invoked again, approximate execution time and priority, which determines its importance level. A real-time task specifies,in addition to above, a deadline by which it must complete execution.*

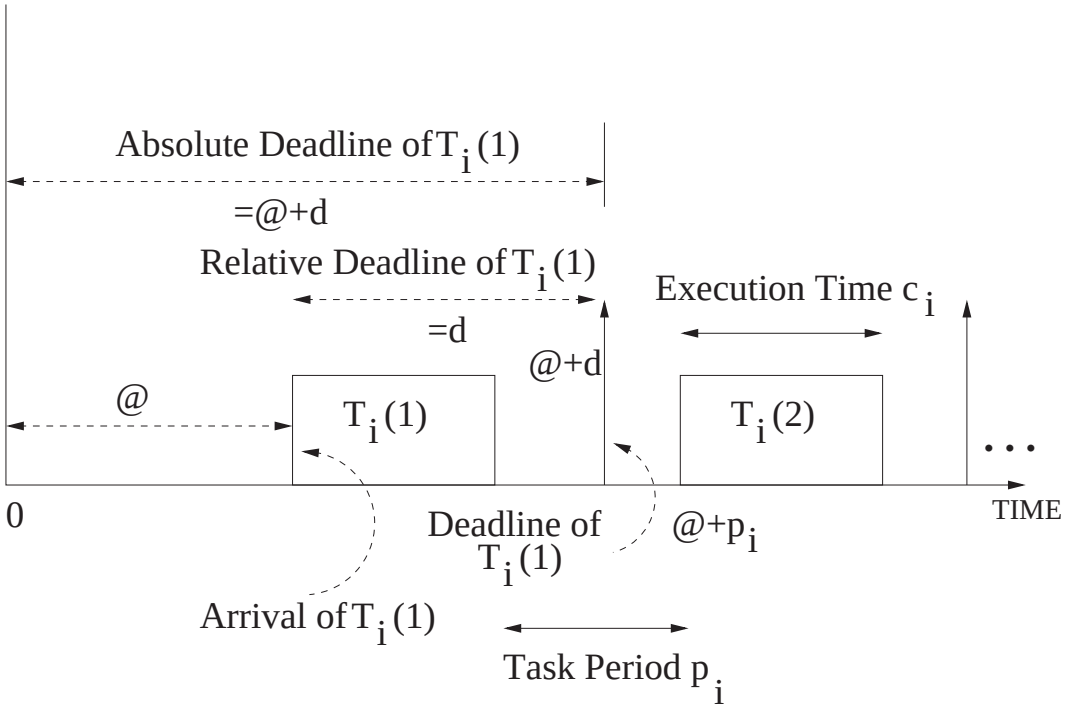


Figure 2.3: Timing Diagram of Periodic Task

The real time tasks can be modeled by parameters listed in Table 2.1. Based on the recurrence of the real time tasks, it is possible to classify them into three main categories: periodic, sporadic and aperiodic tasks.

1. Periodic Tasks: A periodic task is the one that repeats after certain fixed time interval, usually demarcated by clock interrupts. Hence, periodic tasks are sometimes referred to as clock-driven tasks. The fixed time interval after which a tasks repeats is called the period of the tasks. Formally, a periodic task is represented by four tuples (ϕ_i, p_i, c_i, d_i) where ϕ_i is the occurrence of the instance of T_i, p_i is the

Sl.	Parameter	Notation	Description
1.	Arrival Time of a task	A_i	
2.	Absolute deadline of a task	D_i	
3.	Relative Deadline	d_i	
4.	Execution time of a task	C_i	
5.	Maximum execution time	C_i^+	
6.	Minimum execution time	C_i^-	
7.	Completion time	CT_i	
8.	Laxity of a task	L_i	$L_i = D_i - A_i - C_i$
9.	Slack of a task	SL_i	$L_i = SL_i \sum_{i=1}^{N} C_i$
10.	Waiting time	WT_i	
11.	Task arrival rate	AR	
12.	Total number of nodes	M	
13.	Total Load of the system	TL	$TL = AR\frac{\sum_{i=1}^{m} C_i}{m}$
14.	release time of the task	r_i	the release time of the sporadic and periodic task are randomnumbers
15.	Earliest release time	r_i^-	
16.	Latest release time	r_i^+	
17.	Worst case execution time	c_i	

Table 2.1: Parameters for Real Time workload

period of task, c_i is the worst case execution time of task, and d_i is the relative deadline of the task. Periodic tasks can have additional parameters as shown in Table 2.2.

2. Sporadic Tasks: A sporadic task recurs at random instants, A sporadic task T_i can be represented by three tuple (c_i, g_i, d_i) where c_i is the worst case execution time of an instance of the task, g_i denotes the minimum separation between two consecutive instances of the task, d_i is the relative deadline.

3. Aperiodic Task: An aperiodic task is similar to a sporadic task except that the minimum separation g_i between two consecutive instances can be 0.

The basic task model, considered in this thesis work is Ω which is modeled by a set of N periodic tasks T_i:

$$\Omega = \{T_i = (p_i, d_i, c_i) \mid 1 \leq i \leq N\} \tag{2.4}$$

where

p_i is the period of the task. Each task is released every p_i time units. For non-periodic tasks, p_i is represented the minimum (or average) separation time between

Sl.	Parameter	Notation	Description
1.	period of the task	p_i	At all times the period and execution time of periodic tasks is known.
2.	Phase of the task	ϕ_i	$\phi_i = r_{i,1}(J_{i,1})$ i.e. the release time of the first job J_i in task T_i
3.	Utilization of periodic task	u_i	$u_i = \frac{e_i}{p_i}$
4.	Total utilization	U	
5.	The time interval of length H is called hyperperiod of the periodic tasks	H	

Table 2.2: Additional Parameters for periodic tasks

two consecutive releases. The difference in time between the arrivals of two consecutive instances of a periodic task is always fixed and is referred to as period of that task. Although the inter arrival times of instances of a periodic task are fixed, the inter release time may not be.

d_i is the deadline, the period of time after the release time within which the task has to finish, Tasks can have arbitrary deadline.

c_i is the worst-case execution time of the task at each release. In real time systems it is often necessary to determine an upper bound of time in that the program block is executed. Until specified tasks will be represented as T_i

Application program is considered as a set of tasks i.e. $A_i = (T_{A_i,1}, T_{A_i,2}, ...T_{A_i,k})$. Each task consists of subtasks such that $T_{i,1} = st_1^{i,1}, st_2^{i,1}, \ldots, st_n^{i,1}$. The subtasks are considered to be the correct states of a given task. Each task is mapped on a certain node of the distributed application. Hence, the resource requirement of a task

$$T_i = \begin{cases} P_i, .., P_{i+k} & \text{k } \epsilon \text{ (0,1,2, .., n)} \\ ch_j, .., ch_{j+k} & \text{k } \epsilon \text{ (0,1,2, .., m)} \end{cases}$$

The above model has be further enhanced/specialized with the following qualifiers.

2.3.1 Inter-task Dependencies

Tasks may have precedence constraints, which specify in any tasks needs to precede other tasks. If task T_i's output is needed as input by the task T_j, then task T_j is

constrained to be preceded by task T_i. Schedulers that have tom deal with task dependencies usually have to construct a direct acyclic graph (DAG) to represent them. The nodes and edges of a task DAG may have weights associated with them, representing computation and communication costs, respectively. *DAG* of a task is defined as $T = (V, E)$. Vertex represents a subtask and arc represents the connection between connected/dependent subtasks $T_{i,1} = \{st_1^{i,1}, st_2^{i,1}, \ldots, st_n^{i,1}\}$. A task is said to precede another task, if the first task must complete before the second task can start. We consider each Task T_i to consist of a set of subtasks, which have timing and precedence constraint as shown in Figure 2.4. The subtasks st_4^iin a precedence constraint task T_i cannot be executed until the subtasks preceded by it have completed their execution i.e. st_2^i and st_3^i. The subtasks of a task are considered to be the valid states of the given task. The notation used for task and data dependency is listed in Table 2.3.

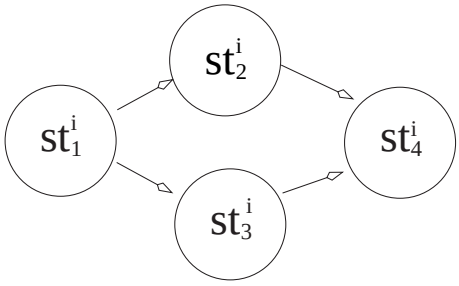


Figure 2.4: Task Graph for with four subtasks

Sl.	Parameter	Notation	Description
1.	Partial order	$<$	called the precedence relation.
2.	immediate predecessor	$T_i < T_k$	T_i is the immediate predecessor of T_k
3.	A precedence graph set	$G = (J, <)$	
4.	If only k out l of its immediate predecessor must be complete before its execution can begin	$k - out - l$	

Table 2.3: Precedence constraint and Data dependency

2.3.2 Priority

Each task has a priority $Pr_{T_i}(index)$ denoted by its sub index, 1 being the highest and N the lowest. Traditionally schedulability techniques use the sub index as the tasks priority. This is correct for a simple task model but when the task model is extended

this might not be the case. For example two tasks sharing the same priority. In this work the relative deadline is considered to be the priority of the task.

For the system considered for study in this thesis, a software based fault tolerant architecture which runs on the CPU of each node has been designed. The main component of the software architecture is a real time kernel which supports time triggered activities. An activity is defined as either the execution of task or as the transmission of a message on the network κ. The computational model can be defined as the sequence of events. Let the initial state of the systems be S_0, and let

$$seq = (e_i, 0 \leq i \leq n) \tag{2.5}$$

be the sequence of events. Suppose that system state when event e_i occurs is S_i. The sequence of events *seq* is a *computation of the system* if the following conditions are satisfied:

1. The event $e_i \epsilon ready(S_i), 0 \leq i \leq n$.
2. $S_{i+1} = next(S_i, e_i), 0 \leq i \leq n$. -

In this model, processes execute simultaneously, the events do occur concurrently. the concurrent execution of events is modeled by non-deterministic interleaving of events. The model also permits the actions of different processes to interleave; any action from any process can occur in a system state provided its enabling condition is satisfied.

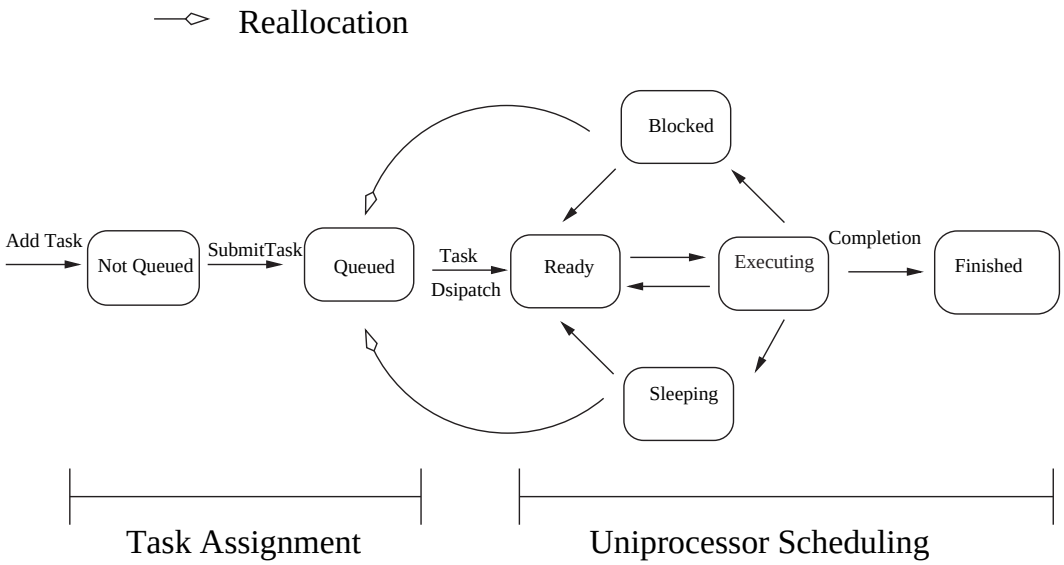


Figure 2.5: Task States in RTDS

Tasks in a Real Time Distributed System can be any one of states as shown in Figure 2.5. The fundamental states are:

- **Not Queued**: Includes all tasks which arrive to the system but are not added to the System queue to be scheduled.

- **Queued**: When the tasks which are added into the systems queue to be scheduled belongs to this state.
- **Ready**: Tasks in this state are the ones assigned/mapped to the processors, according to a scheduling scheme, and are waiting to get their share of resource i.e. processor.
- **Executing**: Tasks which have started executing belong to this state.
- **Finished**: Tasks after completing their execution go to this state.
- **Blocked**: Tasks in this state are preempted or is waiting for something (Example I/O). Such a task will become Ready either if the preemption condition elapses or if the time-out expires. Tasks in the state Blocked cannot run because they are waiting for an event (e.g., a semaphore signal or a message coming in at a mailbox). These tasks can only be made Ready by another task or by an interrupt handler.
- **Sleeping**: The state of a task that is asleep for some duration.

By now we have understood that a Real Time Distributed Computing System (RTDS) includes hardware, software, humans, and the physical world. The four fundamental properties that characterize the RTDS are: functionality, performance, dependability and cost. Real Time Distributed Systems have a common frontier between itself and its environment called the system boundary. The communication in an RTDS includes interaction among its constituent nodes and between itself and its environment. From structural point of view, an RTDS is a set of components bound together in order to interact, where each component is another system. When no further internal structure cannot be discerned, or interest and can be ignored then the component is considered to be atomic.

2.4 Performance Metrics

The real time distributed systems use a variety of metrics depending on the type of real time system they are dealing with. This makes the comparison of schedulers difficult. the requirements of good performance measure has been discussed in [54]. Traditional reliability , availability, and throughput are not suitable performance measures for real time computers[53]. The different types of task characteristics, which occur in practice, cause further difficulty. Scheduling algorithms also vary significantly depending on the type of computer system they are intended for. Finally there can be difference in objectives of scheduling algorithms. The most commonly used performance metrics in Fault tolerant RTDS are (i) Guarantee ratio (ii) Lateness (iii) Makespan (iv) Reliability (v) Availability and (vi) Throughput as listed in Table 2.4

Before we move to next section three essential notions are defined :

Sl.	Parameter	Notation	Description
1.	Guarantee Ratio	GR	$GR = \frac{Total\ number\ of\ task\ guaranteed}{Total\ number\ of\ effective\ task}$
2.	Lateness	$Lateness$	measure how successful we are in meeting the desired deadlines with different schedules $Lateness = CT_i - D_i$
3.	The response time of the set of jobs as a whole.	$Makespan(\partial)$	$\partial = \sum_{i=1}^{N} t_i \left(\sum_{j=1}^{M} P_j \right)^{-1} + \sum_{j=1}^{M} \left(\frac{L_i}{P_i} \right)$
4.	Probability that the system will not undergo failure within any prescribed time interval.	$Reliability$	Reliability of the system over a given period of operation t is given by $\sum_{i \epsilon FAIL} \pi_i(t)$ where the set FAIL consists of all system state that are defined as failures
5.	Fraction of time the system is up	$Availability$	$MTBF = \frac{No.\ of\ hours\ system\ in\ use}{No.\ of\ failures\ encountered}$ $MTTR = \frac{No.\ of\ hours\ spent\ reparing}{No.\ of\ repairs}$ $Availability = \frac{MTBF}{MTBF+MTTR}.100\%$
6.	Average number of instruction per unit time that the system can process.	$Throughput$	If the throughput is x instructions per unit time $x \sum_{i=states}^{1} \pi_i(\prod) + 2x \sum_{i=states+1}^{finalstate} \pi_i(\prod)$

Table 2.4: Performance metrics in Fault Tolerant RTDS

1. Failure characterizes a wrong delivery of the computing system service. The system has an actual behavior, that is not in accordance with the expected behavior, as defined by the specification.

2. Fault is a failure cause. It is often expressed as a non-respect for a property on the designed system structure. A connection of an integrated circuit being broken or a program statement being bad are two examples. To point out the failure origin is sometimes difficult, for instance when detailed knowledge of the structure is not available or when the causes come from outside or are multiple and combined.

3. Error is an intermediate notion. It characterizes the fault effect as an undesirable internal functioning state of the system. A gate output stuck-at 1 or the access to an array element that is out of range are two examples.

A cause and effect relationship exists between fault, error and failure. However, all faults do not mandatory lead to an error, which in turn does not mandatory lead to a failure. For example, a bad statement of a program (that is, a fault) may have no effect (no error) if this statement is not used (dead code due to reuse). The assignment of a value in an array, out of range, is an error. It does not provoke a failure if the crushed value is no longer used by the program execution.

2.5 Faults Classification in RTDS

The service delivered by Real Time Distributed System is its behavior as it is perceived by its user. Since a service is a sequence of the system's external states, a service failure means that at least one (or more) external state of the system deviates from the correct service state. The deviation is called an error. The adjudged or hypothesized cause of an error is called a fault. In most cases a fault first causes an error in the service state of a component that is a part of the internal state of the system and the external state is not immediately affected. For this reason the definition of an error is: the part of the total state of the system that may lead to its subsequent service failure. It is important to note that many errors do not reach the system's external state and cause a failure. A fault is active when it causes an error, otherwise it is dormant. Fault has been formally defined by means of closure, convergence and safety, liveliness in [7, 33] respectively. Situations involving multiple faults and/or failures are frequently encountered. Given a system with defined boundaries, a single fault is a fault caused by one adverse physical event or one harmful human action. Multiple faults are two or more concurrent, overlapping, or sequential single faults whose consequences, i.e., errors, overlap in time, that is, the errors due to these faults are concurrently present in the system. Consideration of multiple faults leads one to distinguish a) independent faults, that are attributed to different causes, and b) related faults, that are attributed to a common cause. Related faults generally cause similar errors.

An RTDS system can fail because of transient overloads caused by excessive stimuli from the environment, or caused by indefinitely blocked programs engaged in resource contention. Furthermore, processors and communication channels can fail, possibly resulting in total system failure. Consequently, design methods for real time distributed systems that are tolerant of component failures and adaptive to environment behaviors while preserving predictability are required. All faults that may affect a real time distributed system during its life are classified according to six basic viewpoints that are shown in Figure 2.6

The fault classification criterion in Real Time Distributed Systems is as follows:

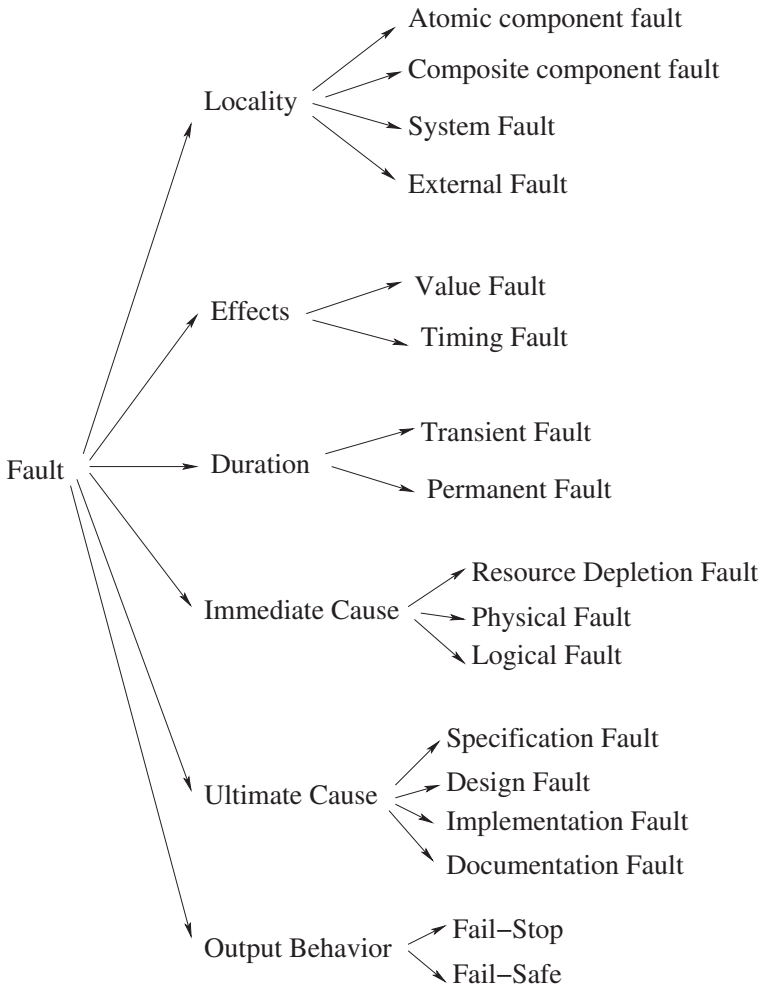


Figure 2.6: Classification of Faults in RTDS

1. Locality with respect to the system boundary faults can be classified as
   - Atomic component fault: Fault in component that cannot be subdivided.
   - Composite component fault: Aggregation of atomic faults.
   - System fault: Fault in System structure rather than system's component.
   - External fault: Fault that arises beyond system boundary ex. user, environment.
2. Effects with respect to the deviation from the viewpoint of the system
   - Value fault: Computation returns result that does not meet system specification.
   - Timing fault: Process or Service not delivered or completed in time.
3. Duration with respect to the time duration for which the fault persists.
   - Transient fault: A server fails but recovers
   - Permanent fault: A server fails and does not recover

4. Immediate Cause when the system violates system specification

    - Resource depletion fault: System unable to receive resource required to perform task.
    - Physical fault: Hardware breaks or a mutation occurs in executable software.
    - Logical fault: System does not respond according to specification.

5. Ultimate Cause during the phase cycle of the system development

    - Specification fault: Improper requirement specification.
    - Design fault: System design does not match the requirement.
    - Implementation fault: System implementation does not actually implement the design.
    - Documentation fault: The documented system does not the real system.

6. Output Behavior according to the state of the system when it encounters a fault.

    - Fail-stop: A unit responds to up to a certain maximum number of failure by simply stopping.
    - Fail-safe: Unit gives out certain result in the light of fault occurrence Ex. Traffic light controller.

2.6 Relationship between Faults, Errors and Failures

The failure modes characterize incorrect service according to four viewpoints: a) the failure domain, b) the detectability of failures, c) the consistency of failures, and d) the consequences of failures on the environment.

A system is said to fail when it cannot meet its promises. In particular, if a distributed system is designed to provide its users with a number of services, the system has failed when one or more of those services cannot be completely provided this is called service failure [22]. An error is a part of a system's state that may lead to a failure. The cause of an error is called a fault. An error is the manifestation in the system of a fault, and a failure is the manifestation on the service of an error, a distinction is made between preventing, removing, and forecasting faults [56]. Deviation of system from its predefined set of valid services leads to a fault [36, 50]. Dependability impairments have also been studied in [69]. Error can be classified according to subsystems or devices which constitute the entire system [104]. Some of the possible errors in an RTDS has been mentioned as in Table 2.6.

The creation and manifestation mechanisms of faults, errors, and failures are illustrated by Figure 2.7, and summarized as follows:

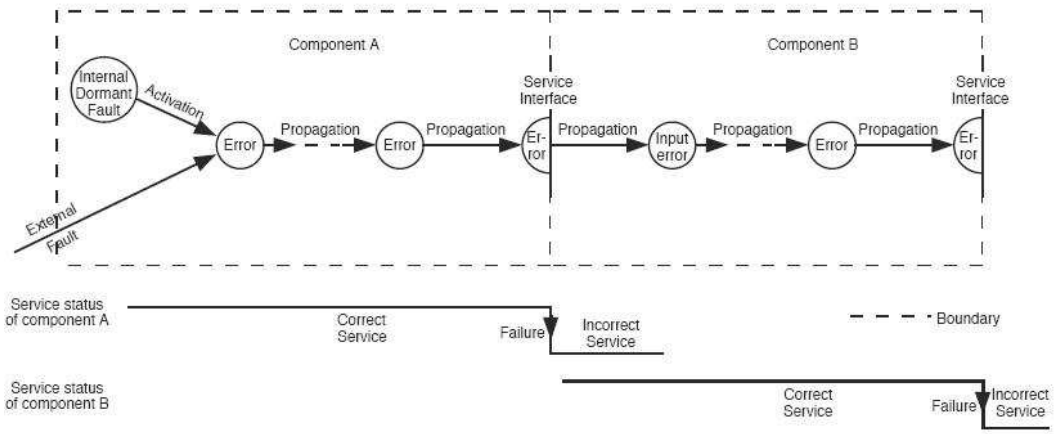


Figure 2.7: Error Propagation

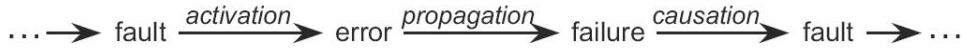


Figure 2.8: Fundamental Chain of dependability threats

1. A fault is active when it produces an error, otherwise it is dormant. An active fault is either a) an internal fault that was previously dormant and that has been activated by the computation process or environmental conditions, or b) an external fault. Fault activation is the application of an input (the activation pattern) to a component that causes a dormant fault to become active. Most internal faults cycle between their dormant and active states.

2. Error propagation within a given component (i.e., internal propagation) is caused by the computation process: an error is successively transformed into other errors. Error propagation from one component (C1) to another component (C2) that receives service from C1 (i.e., external propagation) occurs when, through internal propagation, an error reaches the service interface of component C1. At this time, service delivered by C2 to C1 becomes incorrect, and the ensuing failure of C1 appears as an external fault to C2 and propagates the error into C2.

3. A service failure occurs when an error is propagated to the service interface and causes the service delivered by the system to deviate from correct service. A failure of a component causes a permanent or transient fault in the system that contains the component. Failure of a system causes a permanent or transient external fault for the other system(s) that interact with the given system.

These mechanisms enable the fundamental chain to be completed, as indicated by Figure 2.8. The arrows in this chain express a causality relationship between faults, errors and failures. They should be interpreted generically: by propagation, several errors can be generated before a failure occurs. Propagation, and thus instantiation(s) of

the chain, can occur via the two fundamentals dimensions associated to the definitions of systems given in Section 2.5: interaction and composition

2.7 Dependability Assessment

The ability to deliver service is called dependability. The schema of dependability computing, is mentioned in Table: 2.5. Means to attain dependability has been grouped by researchers into four major categories:

- Fault prevention/avoidance: Fault prevention aims at reducing the creation or occurrence of faults during the computing system life cycle. Means are used during the system design phase. Some of them have an impact on the created system. Others prevent faults occurring during its useful life. These means concern the system modeling tools (including implementation technologies), the system models and the processes used to obtain these models.

- Fault tolerance: Fault tolerance aims at guaranteeing the services delivered by the system despite the presence or appearance of faults. Fault tolerance approaches are divided into two classes:

  - compensation techniques for which the structural redundancy of the system masks the fault presence, and,
  - error detection and recovery techniques, that is, detection and then resumption of the execution either from a safe state or after the operational structure modification (reconfiguration).

  Error recovery techniques are split into two sub-classes:

  - backward recovery aiming at resuming execution in a previously reached safe state;
  - forward recovery aiming at resuming execution in a new safe state.

- Fault removal: Fault removal aims at detecting and eliminating existing faults. Fault removal are older that those on fault prevention. Fault removal techniques are often considered at the end of the model definition, particularly when an operational model of the system is complete.

- Fault evasion: means to estimate the present number, the future incidence, and the likely consequences of faults.

The techniques are used in different steps of system development as shown in figure 2.9 Normally, dependability is defined in statistical terminology, stating the probability that the system is functional and provides the expected service at a specific point in time. This results in common definitions like the well-known mean time to failure (MTTF).

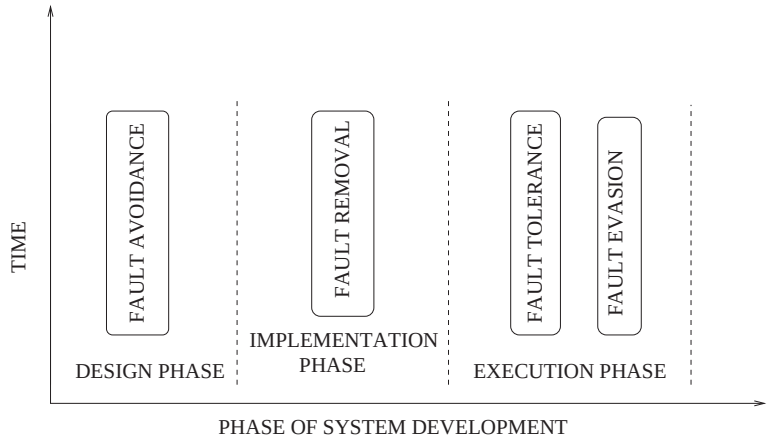


Figure 2.9: Various techniques to achieve dependability in system design phase

Dependability Computing		
Attributes	Threats	Means
Availability	Faults	Fault Tolerance
Reliability	Errors	Fault Removal
Safety	Failures	Fault Forecasting
Confidentiality		
Integrity		
Maintainability		

Table 2.5: Taxonomy of Dependability Computing Schema

While terms like dependability, reliability [77], and availability [16] are important in practical settings. Failures in RTDS can be internal as well as external. Functional

Components	Possible Errors
CPU	CPU related errors, for example, cache parity
Memory	memory ECC errors
Disk	disk, drive, and controller errors
Network	CI bus and CI port errors
Software	software detected errors
Unknown	unknown device or unknown reason errors.

Table 2.6: Error Classification in RTDS

models to failure modes is listed in Table: 2.7. These failures form a hierarchy, with crash fault being simplest and most restrictive type and Byzantine being least restrictive. They can be arranged in an hierarchy as shown in Figure 2.10 [42].

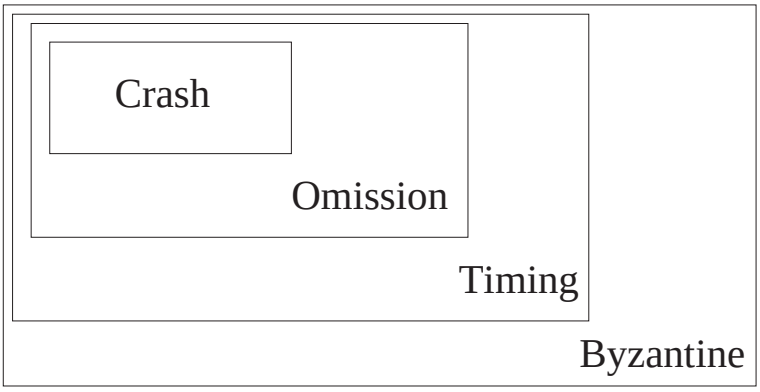


Figure 2.10: Fault Heirarchy

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Byzantine failure	A server may produce arbitrary responses at arbitrary times

Table 2.7: Different type of failures in RTDS

2.8 Applications of Fault Tolerant Computing

Fault tolerant computing are used in critical computation applications where we need real-time control for example; power plant, hospitals, aircraft, weapon, etc. Long-life applications such as unmanned spacecraft need to be highly dependable which can be achieved by incorporating fault tolerant techniques into the system. Similarly, here are high availability applications for example; electronic switching system, OLTP, network switching equipment also need fault tolerant. In many applications maintenance operations are extremely costly, inconvenient, or difficult to perform design the system so unscheduled maintenance can be avoided in telephone switching systems, spacecraft by fault tolerance.

2.9 Fault Tolerant Issues in RTDS

Fault-tolerance in real-time systems is defined informally as the ability of the system to deliver correct results in a timely manner even in the presence of faults during execution time. Dependable real-time systems are being developed in diverse applications including avionics, air-traffic control, plant automation, automotive control, telephone switching, and automatic stock trading. These systems often operate under strict dependability and timing requirements that are imposed due to the interaction with the external environment. Meeting these requirements is complicated by the fact that a real-time system can fail not only because of software or hardware failures, but also because the system is unable to execute its critical functions in time [41].

With modern commercial microprocessors, it is no longer possible to use the traditional hard-real-time fault tolerance strategy of two or more processors operating in hardware lock step with hardware detection of differences in the values computed by the processors. Modern microprocessors contain so much internal concurrency and non-determinism that it is not possible for hardware lock step mechanisms to maintain consistency between the values computed by the processors. Consequently, future fault tolerance mechanisms must be implemented in software [66].

Fault tolerance requires detection of and recovery from faults, which introduces unpredictable delays into the generation of results, in conflict with the predictability required for real-time operation. The greatly improved performance of modern microprocessors mitigates the recovery delays to some extent, but at a cost. Scheduling Given goals, how should tasks be scheduled? Periodic, aperiodic and completely ad-hoc tasks What should we do if a system misses its goals? How can we make components highly predictable in terms of their real-time performance profile?

In fault-tolerant real time distributed systems, detection of fault and its recovery should be executed in timely manner so that in spite of fault occurrences the intended output of real-time computations always take place on time. For a fault tolerant technique detection latency and recovery time are important performance metrics because they contribute to server down-time. A fault tolerant technique can be useful, in RTDS if its fault detection latency and recovery time are tightly bounded. When this is not feasible, the system must attempt the fault tolerance actions that lead to the least damages to the application's mission and the system's users.

Fast reconfiguration, which includes functional amputation of faulty components and redistribution of tasks to existing, newly incorporated, and repaired nodes. Another major issue in real-time fault-tolerant DC is scalability. Wide area network infrastructure is increasingly used in new applications. To cope with this trend, the research community must enhance the scalability of network surveillance techniques and recovery techniques.

Also, currently the main challenge in development of the real-time fault-tolerant DC technology appears to be integration. There are two major types of integration that

must be developed: real-time fault-tolerant computing stations and network surveillance and reconfiguration; and fault detection and replication principles and object-oriented real-time DC structuring techniques. Issues in FTRTDS is further explained in [50].

2.10 Fault Tolerance Techniques

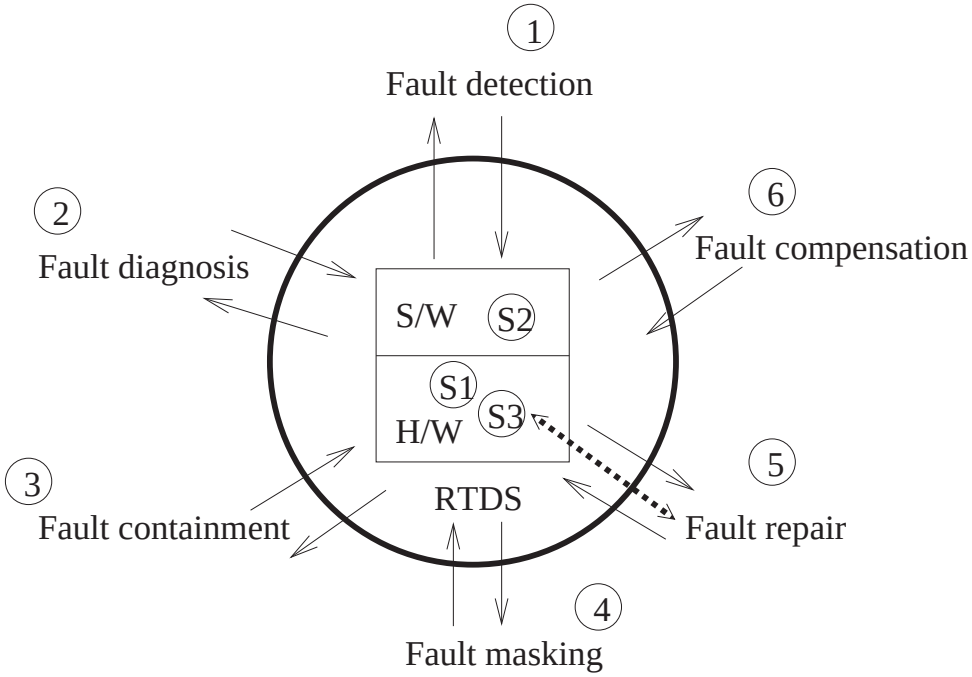


Figure 2.11: Life Cycle of fault handling

Fault tolerance mechanism involves no of following steps, each step is associated with specific function hence they can be applied independently in the process of fault handling and the life cycle of fault handling is illustrated in Figure 2.11

Fault Detection : One of the most important aspects of fault handling in RTDS is detecting a fault immediately and isolating it to the appropriate unit as quickly as possible [47]. In distributed system there is no central points for lookout from which the entire system can be observed at once [9] hence fault detection remains a key issue in distributed system. [4] reveals the impact of faster fault detectors in Real Time System. Design goals and a architecture for fault detection in grid has been discussed in [103, 42]. Commonly used fault detection techniques are Consensus, Deviation Alarm and Testing.

Fault Diagnosis : Figure out where the fault is and what caused the fault for example Voter in TMR can indicate which module failed and Pinpoint can identify failed components

Fault Containment/Fault Isolation : If a unit is actually faulty, many fault triggers will be generated for that unit. The main objective of fault isolation is to correlate the fault triggers and identify the faulty unit and then confine the fault to prevent infection i.e. prevent it to propagate from its point of origin.

Fault Masking : Ensuring that only correct value get passed to the system boundary inspite of a failed component.

Fault Repair/Recovery : A process in which faults are removed from the system. Fault Repair/Recovery Techniques can be of the following types include Checkpoint [74, 93] and Rollback. A reconfiguration of stateful processes is discussed in [59]

Fault Compensation : If a fault occurs and is confined to a subsystem, it may be necessary for the system to provide a response to compensate for output of the faulty subsystem.

During fault tolerance all of the above steps may not be involved.

2.10.1 Redundancy

Redundancy is the heart of fault tolerance there are four type of redundancy:

1. Hardware Redundancy: Based on replication of physical components.
2. Software Redundancy: The system is provided with different software versions of tasks, preferably written independently by different teams.
3. Time Redundancy: Based on multiple executions on the same hardware in different times.
4. Information Redundancy: Based on coding data in such a way that a certain number of bit errors can be detected and/or corrected.

2.10.2 Checkpointing

In checkpoint-based methods, the state of the computation as a checkpoint is periodically saved to stable storage, which is not subject to failures. When a failure occurs, the computation is restarted from one of these previously saved states. According to the type of coordination between different processes while taking checkpoints, checkpoint-based methods can be broadly classified into three categories: uncoordinated checkpointing, coordinated checkpointing and communication-induced checkpointing.

1. Uncoordinated: In uncoordinated checkpointing, each process independently saves its state. During restart, these processes search the set of saved checkpoints for a consistent state from which the execution can resume. The main advantage

of this scheme is that a checkpoint can take place when it is most convenient. For efficiency, a process may perform checkpoints when the state of the process is small [23]. However, uncoordinated checkpointing is susceptible to rollback propagation, the domino effect [19] which could possibly cause the system to roll-back to the beginning of the computation resulting in the waste of a large amount of useful work. Rollback propagations also make it necessary for each processor to store multiple checkpoints, potentially leading to a large storage overhead. Due to the potentially unbounded cost of rollback, we consider uncoordinated checkpointing unsuitable for our requirements.

2. Coordinated: Coordinated checkpointing requires processes to coordinate their checkpoints in order to form a consistent global state. Coordinated checkpointing simplifies recovery from failure because it does not suffer from rollback propagations. It also minimizes storage overhead since only one checkpoint is needed. Coordinated checkpoint schemes suffer from the large latency involved in saving the checkpoints since a consistent checkpoint needs to be determined before the checkpoints can be written to stable storage. In most cases, a global synchronization is needed to determine such a consistent global state.

3. Communication induced checkpointing (CIC): allows processes in a distributed computation to take independent checkpoints and to avoid the domino effect. CIC protocols are believed to have several advantages over other styles of rollback recovery. For instance, they allow processes considerable autonomy in deciding when to take checkpoints. A process can thus take a checkpoint at times when saving the state would incur a small overhead . CIC protocols also are believed to scale up well with a larger number of processes since they do not require the processes to participate in a global checkpoint. CIC protocols do not scale well with a larger number of processes. CIC protocols seem to perform best when the communication load is low and the pattern is random. Regular, heavy load communication patterns seem to fare worse [5]

Checkpointing in a real-time system has some important differences compared to that employed in a traditional distributed and database system. Normally a real-time task does not require an audit trail (except in the case of task which access global data), since all the data are assumed to be local upon initiation of the task. The saved state of the task, which consists of values of data variables and contents of system registers, is called a checkpoint [79]. To ensure correctness of a checkpoint with certain failure models, an acceptance test must be executed before saving the necessary data. Basically an acceptance test is checking a condition which is expected to happen if the program has successfully executed. In principle it is not intended to guarantee absolute correctness of the results and hence could be employed at diverse levels ranging from a simple reasonableness check to exhaustive verification of output parameters. Again this involves a tradeoff between how comprehensive the acceptance tests are, versus

the associated design and run-time costs. These requirements are standard in fault tolerant systems.

2.10.3 Hardware Fault Tolerance

Hardware based fault tolerance is achieved by physical replication of hardware. The majority of fault-tolerant designs have been directed toward building computers that automatically recover from random faults occurring in hardware components. The techniques employed to do this generally involve partitioning a computing system into modules that act as fault-containment regions. Each module is backed up with protective redundancy so that, if the module fails, others can assume its function. Special mechanisms are added to detect errors and implement recovery.

2.10.3.1 General Approaches to Hardware Fault Recovery

Two general approaches to hardware fault recovery have been used: 1) fault masking, and 2) dynamic recovery.

1. Fault masking is a structural redundancy technique that completely masks faults within a set of redundant modules. A number of identical modules execute the same functions, and their outputs are voted to remove errors created by a faulty module. Triple modular redundancy (TMR) is a commonly used form of fault masking in which the circuitry is triplicated and voted. The voting circuitry can also be triplicated so that individual voter failures can also be corrected by the voting process. A TMR system fails whenever two modules in a redundant triplet create errors so that the vote is no longer valid. Hybrid redundancy is an extension of TMR in which the triplicated modules are backed up with additional spares, which are used to replace faulty modules – allowing more faults to be tolerated. Voted systems require more than three times as much hardware as nonredundant systems, but they have the advantage that computations can continue without interruption when a fault occurs, allowing existing operating systems to be used.

2. Dynamic recovery is required when only one copy of a computation is running at a time (or in some cases two unchecked copies), and it involves automated self-repair. As in fault masking, the computing system is partitioned into modules backed up by spares as protective redundancy. In the case of dynamic recovery however, special mechanisms are required to detect faults in the modules, switch out a faulty module, switch in a spare, and instigate those software actions (rollback, initialization, retry, restart) necessary to restore and continue the computation. In single computers special hardware is required along with software to do this, while in multicomputers the function is often managed by the other processors. Dynamic recovery is generally more hardware-efficient than voted systems, and it is therefore the approach of choice in resource-constrained

(e.g., low-power) systems, and especially in high performance scalable systems in which the amount of hardware resources devoted to active computing must be maximized. Its disadvantage is that computational delays occur during fault recovery, fault coverage is often lower, and specialized operating systems may be required.

2.10.3.2 Reliability Wise Component Configurations

Hardware fault tolerance can also be achieved by reliability wise Component Configurations. The configuration can be as simple as units arranged in a pure series or parallel configuration. There can also be systems of combined series/parallel configurations or complex systems that cannot be decomposed into groups of series and parallel configurations. The configuration types included here are:

1. Series Configuration: In a series configuration, a failure of any component results in failure for the entire system. In most cases when considering complete systems at their basic subsystem level, it is found that these are arranged reliability-wise in a series configuration. For example, a personal computer may consist of four basic subsystems: the motherboard, the hard drive, the power supply and the processor. These are reliability-wise in series and a failure of any of these subsystems will cause a system failure. In other words, all of the units in a series system must succeed for the system to succeed.

   The reliability of the system is the probability that unit 1 succeeds and unit 2 succeeds and all of the other units in the system succeed. So, all n units must succeed for the system to succeed. The reliability of the system R_s is then given by:

$$\begin{aligned} R_s &= P(X_1 \cap X_2 \cap \ldots \cap X_n) \\ &= P(X_1)P(X_2|X_1)P(X_3|X_1X_2)\ldots P(X_n|X_1X_2\ldots X_{n-1}) \end{aligned} \tag{2.6}$$

   where:

   - R_s = reliability of the system.
   - X_i = event of unit i being operational.
   - $P(X_i)$ = probability that unit i is operational.

   In the case where the failure of a component affects the failure rates of other components (i.e. the life distribution characteristics of the other components change when one fails), then the conditional probabilities in Equation(2.6) must be considered. However, in the case of independent components, Equation(2.6) becomes:

$$R_s = P(X_1)P(X_2)P(X_3)\ldots P(X_n) \tag{2.7}$$

or

$$R_s = \prod_{i=1}^{n} P(X_i) \tag{2.8}$$

or, in terms of individual component reliability:

$$R_s = \prod_{i=1}^{n} R_i \tag{2.9}$$

In other words, for a pure series system, the system reliability is equal to the product of the reliabilities of its constituent components.

2. Simple Parallel Configuration: In a simple parallel system at least one of the units must succeed for the system to succeed. Units in parallel are also referred to as redundant units. Redundancy is a very important aspect of system design and reliability in that adding redundancy is one of several methods of improving system reliability. It is widely used in the aerospace industry and generally used in mission critical systems. Other example applications include the RAID computer hard drive systems, brake systems and support cables in bridges.

   The probability of failure, or unreliability, for a system with n statistically independent parallel components is the probability that unit 1 fails and unit 2 fails and all of the other units in the system fail. So in a parallel system, all n units must fail for the system to fail. Put another way, if unit 1 succeeds or unit 2 succeeds or any of the n units succeeds, then the system succeeds. The unreliability of the system is then given by:

$$\begin{aligned} Q_s &= P(X_1 \cap X_2 \cap \ldots \cap X_n) \\ &= P(X_1)P(X_2|X_1)P(X_3|X_1X_2)\ldots P(X_n|X_1X_2\ldots X_{n-1}) \end{aligned} \tag{2.10}$$

   where:

   - Q_s = reliability of the system.
   - X_i = event of unit i being operational.
   - $P(X_i)$ = probability that unit i is operational.

   In the case where the failure of a component affects the failure rates of other components, then the conditional probabilities in Equation(2.10) must be considered. However, in the case of independent components, Equation (2.10) becomes:

$$Q_s = P(X_1)P(X_2)P(X_3)\ldots P(X_n) \tag{2.11}$$

   Or:

$$Q_s = \prod_{i=1}^{n} P(X_i) \tag{2.12}$$

Or, in terms of component unreliability:

$$Q_s = \prod_{i=1}^{n} Q_i \tag{2.13}$$

Observe the contrast with the series system, in which the system reliability was the product of the component reliabilities; whereas the parallel system has the overall system unreliability as the product of the component unreliabilities.
The reliability of the parallel system is then given by:

$$\begin{aligned} R_s &= 1 - Q_s = 1 - (Q_1 * Q_2 \cdots * Q_n) \\ &= 1 - [(1 - R_1) * (1 - R_2) * \cdots * (1 - R_n))] \\ &= 1 - \prod_{i=1}^{n} (1 - R_i) \end{aligned} \tag{2.14}$$

3. Series and parallel configuration: While many smaller systems can be accurately represented by either a simple series or parallel configuration, there may be larger systems that involve both series and parallel configurations in the overall system. Such systems can be analyzed by calculating the reliabilities for the individual series and parallel sections and then combining them in the appropriate manner.

4. k-out-of-n Systems: The k-out-of-n configuration is a special case of parallel redundancy. This type of configuration requires that at least k components succeed out of the total n parallel components for the system to succeed. For example, consider an airplane that has four engines. Furthermore, suppose that the design of the aircraft is such that at least two engines are required to function for the aircraft to remain airborne. This means that the engines are reliability-wise in a k-out-of-n configuration, where k = 2 and n = 4. More specifically, they are in a 2-out-of-4 configuration.

   As the number of units required to keep the system functioning approaches the total number of units is the system, the system's behavior tends towards that of a series system. If the number of units required is equal to the number of units in the system, it is a series system. In other words, a series system of statistically independent components is an n-out-of-n system and a parallel system of statistically independent components is a 1-out-of-n system.

Reliability of k-out-of-n Independent and Identical Components : The simplest case of components in a k-out-of-n configuration is when the components are independent and identical. In other words, all the components have the same failure distribution and whenever a failure occurs, the remaining components are not affected. In this case, the reliability of the system with such a configuration

can be evaluated using the binomial distribution, or:

$$R_s(k, n, R) = \sum_{r=k}^{n} \binom{n}{r} R^r (1 - R)^{n-r} \tag{2.15}$$

where:

- n is the total number of units in parallel.
- k is the minimum number of units required for system success.
- R is the reliability of each unit.

5. Duplex Systems: Duplex Systems consisting of two nodes and one comparator for a 2-out-of-2 decision among the results are commonly applied as shown in Figure 2.12. It only needs a single processor, which executes both software variants sequentially. Usual absolute tests such as parity checks, instruction validity checks, address range checks and processor self-tests are performed. The final comparison of results is called a relative test.

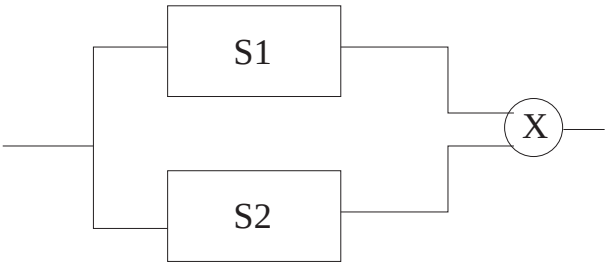


Figure 2.12: Duplex Systems

2.10.3.3 Processor Level Fault Tolerance Technique

Duplication of the processing core is cost intensive and should be avoided for large multiprocessors systems or large multicomputers. Watchdog Processors and Simultaneous multi threading offers a solution to these problems.

Watchdog processors : Majority of processor malfunction results in an incorrect program execution sequence. Thus, the checking of the program control flow is of primary importance. A watchdog processor [75] is a coprocessor concurrently monitoring the program control flow either by observing the instruction fetch on the CPU bus or by checking symbolic labels explicitly sent from the main program. The traditional watchdog processor is designed as add-on to a conventional CPU. Thus, the overhead is of the same order of magnitude as the CPU itself. Another insufficiency of traditional watchdogs is their inability to monitor the cooperation between tasks in a multitasking system and the interactions of the different processors in a multiprocessor.

Simultaneous Multithreading : Simultaneous Multithreading (SMT) is a technique that allows fine-grained resource sharing among multiple threads in a dynamically scheduled superscalar processor. An SMT processor extends a standard superscalar pipeline to execute instructions from multiple threads, possibly in the same cycle. Hence provides transient fault coverage with significantly higher performance.

2.10.4 Software Fault Tolerance

Software fault tolerance is the ability for software to detect and recover from a fault that is happening or has already happened in either the software or hardware in the system in which the software is running in order to provide service in accordance with the specification. Software fault tolerance is not a solution unto itself however, and it is important to realize that software fault tolerance is just one piece necessary to create the next generation of reliable systems.

In order to adequately understand software fault tolerance it is important to understand the nature of the problem that software fault tolerance is supposed to solve. Current software fault tolerance methods are based on traditional hardware fault tolerance.

2.10.4.1 Recovery Block Approach

The recovery block operates with an adjudicator which confirms the results of various implementations of the same algorithm. In a system with recovery blocks, the system view is broken down into fault recoverable blocks. The entire system is constructed of these fault tolerant blocks. Each block contains at least a primary, secondary, and exceptional case code along with an adjudicator. The adjudicator is the component which determines the correctness of the various blocks to try. The adjudicator should be kept somewhat simple in order to maintain execution speed and aide in correctness. Upon first entering a unit, the adjudicator first executes the primary alternate. (There may be N alternates in a unit which the adjudicator may try.) If the adjudicator determines that the primary block failed, it then tries to roll back the state of the system and tries the secondary alternate. If the adjudicator does not accept the results of any of the alternates, it then invokes the exception handler, which then indicates the fact that the software could not perform the requested operation.

Recovery block operation still has the same dependency which most software fault tolerance systems have: design diversity. The recovery block method increases the pressure on the specification to be specific enough to create different multiple alternatives that are functionally the same. This issue is further discussed in the context of the N-version method.

The recovery block system is also complicated by the fact that it requires the ability to roll back the state of the system from trying an alternate. This may be accomplished in a variety of ways, including hardware support for these operations. This try and

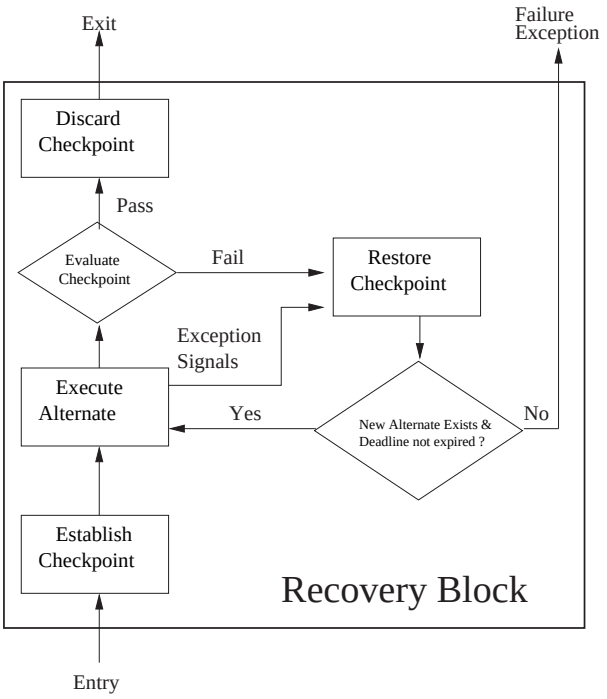


Figure 2.13: Operation of the Recovery Block

rollback ability has the effect of making the software to appear extremely transactional, in which only after a transaction is accepted is it committed to the system. There are advantages to a system built with a transactional nature, the largest of which is the difficult nature of getting such a system into an incorrect or unstable state. This property, in combination with checkpointing and recovery may aide in constructing a distributed hardware fault tolerant system. The operation of the recovery block has been depicted in Figure 2.13.

2.10.4.2 Distributed Recovery Block

The distributed recovery block (DRB) was formulated by K.H. Kim and H. 0. Welch as a means of integrating hardware and software fault tolerance in a single structure for real-time applications. The DRB scheme is capable of effecting forward recovery while handling both hardware and software faults in a uniform manner. The DRB approach combines distributed processing and recovery block concepts [72]. As the name suggests, it is a modification of the (standard) recovery block structure described in the previous section in that it consists of a primary module (e.g., routine), one or more alternate modules, and acceptance tests for both logic and time. The difference between the DRB and the (standard) recovery block is that the primary and alternate modules are both replicated and are resident on two computing nodes which are interconnected by a network. Both computing nodes receive the same input data simultaneously from the previous computing station and compute their modules concurrently.

There are two replicated network nodes designated as primary node and backup

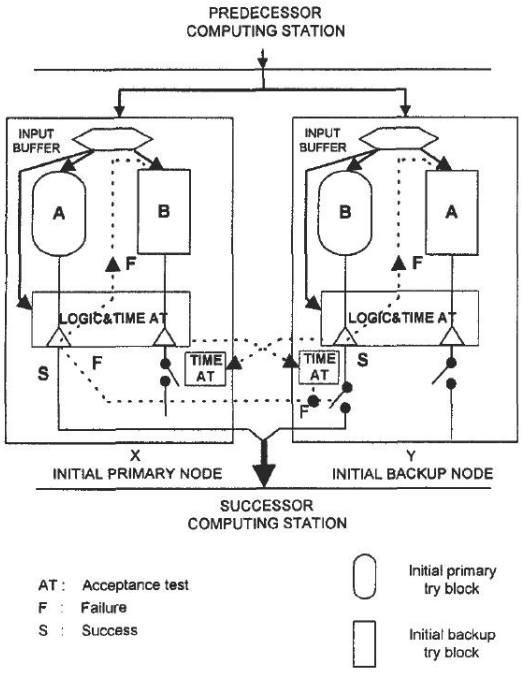


Figure 2.14: Distributed Recovery Block

node. Each of these has the primary and alternate versions of the program resident as well as an acceptance test. The acceptance test unit, itself, has two parts the timer, to determine if the execution of the try block has taken place within an acceptable time since the beginning of the execution by the try block, and the logic acceptance test whose function is to determine if the output of the try block is acceptable or not. The watchdog timer is turned on as soon as an input data set is received as shown in Figure 2.14.

In Kim's terminology, the primary module is referred to as the primary try block; the alternate module is referred to as the alternate try block. In the DRB system, one processor executes the primary try block while the other processor executes the alternate try block. In the fault-free circumstances, the primary node runs the primary try block whereas the backup node runs the alternate try block concurrently. Both will pass the acceptance test and update their local database. Upon success of the primary node, it informs the backup node which updates its own database with its own result. Thereafter, only the primary node sends its output to the successor computing station. In the event of a failure of the primary try block (i.e., routine) as detected by the acceptance test, the primary node informs the backup node of the failure. As soon as the backup node receives notice, it assumes the role of the primary node. Because it has been executing the alternate module concurrently, a result will generally be immediately available for output. Thus the recovery time for this type of failure is much shorter than if both try blocks were running on the same node.

2.10.4.3 N-version Programming

The N-version programming concept attempts to parallel the traditional hardware fault tolerance concept of N-way redundant hardware. In an N-version software system, each module is made with up to N different implementations. Each variant accomplishes the same task, but hopefully in a different way. Each version then submits its answer to voter or decider which determines the correct answer, (hopefully, all versions were the same and correct,) and returns that as the result of the module as shown in Figure 2.15. This system can hopefully overcome the design faults present in most software by relying upon the design diversity concept. An important distinction in N-version software is the fact that the system could include multiple types of hardware using multiple versions of software. The goal is to increase the diversity in order to avoid common mode failures. Using N-version software, it is encouraged that each different version be implemented in as diverse a manner as possible, including different tool sets, different programming languages, and possibly different environments. The various development groups must have as little interaction related to the programming between them as possible. N-version software can only be successful and successfully tolerate faults if the required design diversity is met. The dependence on appropriate

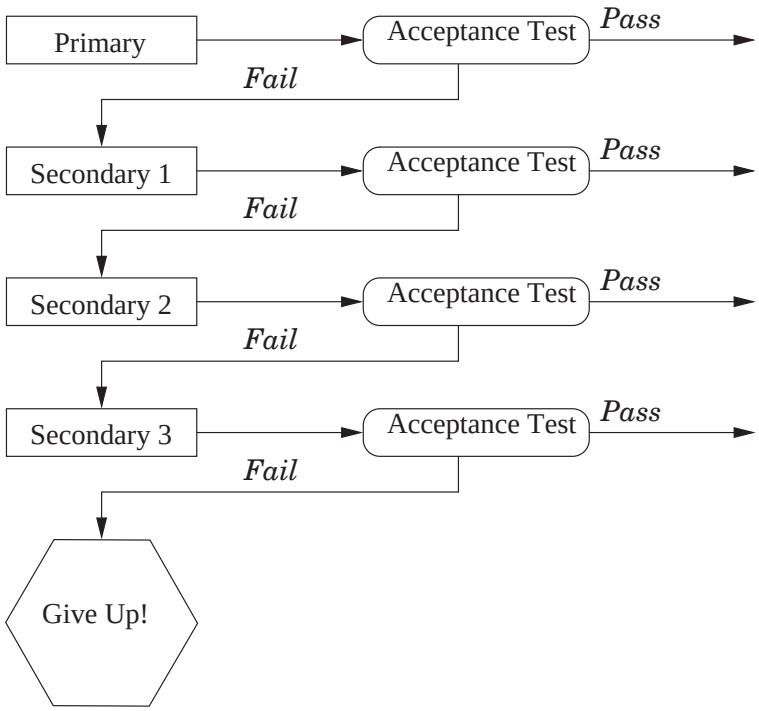


Figure 2.15: N-Version Programming

specifications in N-version software, (and recovery blocks,) can not be stressed enough. The delicate balance required by the N-version software method requires that a specification be specific enough so that the various versions are completely inter-operable, so that a software decider may choose equally between them, but cannot be so limiting that the software programmers do not have enough freedom to create diverse

designs. The flexibility in the specification to encourage design diversity, yet maintain the compatibility between versions is a difficult task, however, most current software fault tolerance methods rely on this delicate balance in the specification.

The N-version method presents the possibility of various faults being generated, but successfully masked and ignored within the system. It is important, however, to detect and correct these faults before they become errors. First, the classification of faults applied to N-version software method: if only a single version in an N-version system, the error is classified as a simplex fault. If M versions within an N-version system have faults, the the fault is declared to be an M-plex fault. M-plex faults are further classified into two classes of faults of related and independent types. Detecting, classifying, and correcting faults is an important task in any fault tolerant system for long term correct operation.

The differences between the recovery block method and the N-version method are not too numerous, but they are important. In traditional recovery blocks, each alternative would be executed serially until an acceptable solution is found as determined by the adjudicator. The recovery block method has been extended to include concurrent execution of the various alternatives. The N-version method has always been designed to be implemented using N-way hardware concurrently. In a serial retry system, the cost in time of trying multiple alternatives may be too expensive, especially for a real-time system. Conversely, concurrent systems require the expense of N-way hardware and a communications network to connect them. Another important difference in the two methods is the difference between an adjudicator and the decider. The recovery block method requires that each module build a specific adjudicator; in the N-version method, a single decider may be used. The recovery block method, assuming that the programmer can create a sufficiently simple adjudicator, will create a system which is difficult to enter into an incorrect state. The engineering tradeoffs, especially monetary costs, involved with developing either type of system have their advantages and disadvantages, and it is important for the engineer to explore the space to decide on what the best solution for his project is.

2.10.4.4 Exception Handling

There are many types of errors or exceptional situations that a program may have to deal with. An exception is a class of computational states that requires an extraordinary computation. It is not possible to give a precise definition of when a computational state should be classified as an exception occurrence; this is a decision for the programmer. In practice, most people have a good feeling of what is the main computation and what are exceptional situations. The exceptional situations are all those situations that imply that the main computation fails.

A program must be able to deal with exceptions. A good design rule is to list explicitly the situations that may cause a program to break down . Many programming languages have special constructs for describing exception handling. Exception

indicates that something happened during execution that needs attention Control is transferred to an exception-handler - routine which takes appropriate action Example: When executing y=a*b, if overflow, result incorrect - signal an exception Effective exception-handling can make a significant improvement to system fault tolerance Over half of code lines in many programs are devoted to exception-handling Exceptions can be used to deal with

- domain or range failure
- out-of-ordinary event (not failure) needing special attention
- timing failure

2.10.4.5 Fault Tolerant Remote Procedure Calls

Remote Procedure Call extends the procedure call mechanism to the distributed environment by allowing a procedure to reside in another node. When a remote call is invoked, the calling environment (known as the caller or the client) is suspended and the parameters are passed across the network to the remote procedure where the execution begins. When the callee completes its execution, the result is passed back through the network to the caller, whose execution is then resumed.

Fault tolerance is provided by having copies of a procedure reside on multiple nodes. Each copy is known as an incarnation. The incarnation are organized in a linear chain. For the ith incarnation the (i+1)st incarnation forms its backup. A service request is made to the primary incarnation, which is the first copy in the chain that has not failed. The primary calee then propagates the call to its backup, which inturn sends the call to its own backup. In this manner, all the incarnation are invoked. The result of the call is returned to the client by the primary calee. If the primary calee fails, its backup incarnation assumes the role of the primary and replies to the client.

Primary Backup Approach : In the PB approach, two versions of a task are executed on two different processors, and an acceptance test (AT) is used to check the result. Three different PB based fault-tolerant approaches have been identified [34]. The two most popular PB approaches are the Primary-Secondary Exclusive (PS-EXCL) and the CONCURrent approaches. PS-EXCL is the most widely used PB approach where the primary and backup versions of the tasks are excluded in space (processor) and time. CONCUR proposes a concurrent execution of the primary and backup versions of each task. This approach obviously involves unnecessary use of resources if faults rarely occur. A third approach is possible, namely, OVERLAP. This approach is a combination of PS-EXCL and CONCUR and is flexible enough to exploit their advantages according to the system parameters.

The Circus Approach : A mechanism for constructing highly available distributed programs. Replicated procedure call combines remote procedure call with replication of program modules for fault tolerance [21]. Remote procedure call allows program

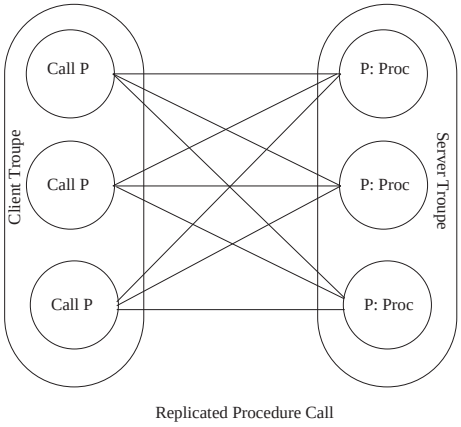


Figure 2.16: Circus: Combines Remote Procedure Call With Replication

modules to be located on different machines. Replicated procedure call generalizes this by allowing modules to be replicated any number of times. The set of replicas of a module is called a troupe. When a client troupe makes a replicated procedure call to a server troupe, each member of the server troupe performs the requested procedure exactly once, and each member of the client troupe receives the results, as shown in Figure 2.16. A distributed program constructed from troupes will continue to function as long as at least one member of each troupe survives.

2.10.5 Network Availability and Fault Tolerance

The majority of communications applications, from cellular telephone conversations to credit card transactions, assume the availability of a reliable network. At this level, data are expected to traverse the network and to arrive intact at their destination. The physical systems that compose a network, on the other hand, are subjected to a wide range of problems, ranging from signal distortion to component failures. Similarly, the software that supports the high-level semantic interface often contains unknown bugs and other latent reliability problems. Redundancy underlies all approaches to fault tolerance.

A wide variety of approaches have been employed for detection of network failures. In electronic networks with binary voltage encodings (e.g., RS-232), two non-zero voltages are chosen for signaling. A voltage of zero thus implies a dead line or terminal. Similarly, electronic networks based on carrier modulation infer failures from the absence of a carrier. Shared segments such as Ethernet have been more problematic, as individual nodes cannot be expected to drive the segment continuously. In such networks, many failures must be detected by higher levels in the protocol stack.

Fault tolerant schemes in networks can be studied under following categories;

1. Path-based Schemes: Protection schemes, in which recovery routes are pre-planned, generally offer better recovery speeds than restoration approaches, which search for new routes dynamically in response to a failure and generally involve software processing. Path protection can itself be divided into several categories: one-plus-one (written 1+1), one-for-one (written 1:1), and one-for-N (1:N). All forms of protection require adequate spatial redundancy in the network topology to allow advance selection of two disjoint routes for each circuit.

2. Link and Node-Based Schemes: Link and node protection can be viewed as a compromise between live and event-triggered path protection. these approaches are independent of traffic patterns, and can be pre-planned once to support arbitrary dynamic traffic loads. Path protection does not provide this feature; new protection capacity may be necessary to support additional circuits, and routes chosen without knowledge of the entire traffic load, as is necessary when allocating routes online, are often suboptimal. This benefit makes link and node restoration particularly attractive at lower layers, at which network management at any given point in the network may not be aware of the origination and destination, or of the format of all the traffic being carried at that location.

3. Rings: Rings have emerged as one of the most important architectural building blocks for backbone networks in the MAN and WAN arenas. While ring networks can support both path-based and link or node-based schemes for reliability, rings merit a separate discussion because of the practical importance of the ring architecture and of its special properties.

4. Mesh Networks: Ring-based architectures may be more expensive than meshes and as nodes are added, or networks are interconnected, ring-based structures may be difficult to preserve, thus limiting their scalability. However, rings are not necessary to construct fault tolerant networks. Mesh-based topologies can also provide redundancy.

5. Packet-Based Approaches: Fault tolerance in packet-switched networks relies on a combination of physical layer notification and re-routing by higher-level routing protocols. The Border Gateway Protocol (BGP), a peer protocol to the Internet Protocol (IP), defines the rules for advertising and selecting routes to networks in the Internet. More specifically, it defines a homogeneous set of rules for interactions between Autonomous Systems, networks controlled by a single administrative entity. With each AS, administrators are free to select whatever routing protocol suits their fancy, but AS's must interact with each other in a standard way, as defined by BGP. BGP explicitly propagates failure information in the form of withdrawn routes, which cancel previously advertised routes to specific networks.

6. High-Speed LAN's: Computing power of PCs continues is growing rapidly and networking computing is fast gaining popularity. This has led to requirements such as centralized server farms, power workgroup and high speed LAN's. High speed LAN's provide more speed and shared bandwidth. It has moved from conventional data to integrated data i.e. conventional and audio/video.

2.11 Related Work

Software based fault tolerant application using a single version scheme (SVS) is described in [95]. A successful fault tolerant primary/backup algorithm with the dynamic EDF algorithm for multiprocessors running in parallel and executing real-time applications [68, 34]. An idea to Biologically Inspired Fault-Tolerant Computer Systems has been proposed in [110]. A layered approach for software fault-tolerance in distributed hard real time systems is discussed in [105]. One of the major technique for achieving fault tolerance is replication but the level of replication is chosen depending on the desired fault tolerance required [100] discusses a replication control mechanism in distributed real time database system. A middleware based MEAD infrastructure [13] aims to provide a reusable, resource-aware real-time support to applications to protect against crash, communication, partitioning and timing faults. Other middleware techniques are discussed in [48]. [52] discusses about fault tolerant computing with more stress on fault tolerant network for distributed systems. Fault tolerance is not a property which can be pursued simplistically in an all-or-nothing fashion. Five different categories of achievable fault tolerance in terms of benefits to the applications can be recognized [49]. Fault tolerant techniques implemented by means of scheduling are discussed in [31, 82, 11, 65, 30, 103]. A fault tolerant communication for RTDS is discussed in [115]. A new technique and related work on Fault tolerance in grid environment is mentioned in [60]. Several architectures for tolerating both hardware and software faults in a given application have been defined assuming a dynamic and distributed computing environment [12, 17, 57, 80].

2.12 Impact of fault on RTDS

High performance computer systems, including real time distributed system encounters an increasing numbers of failure in both atomic and composite components. The end result is that long running, distributed applications are interrupted by failures with increasing frequency. Additionally, when an application does fail, the cost is more higher since more computation is lost. Moreover, Real time systems are usually required to provide an absolute guarantee that all tasks will always complete by their deadlines, especially mission critical real time systems. In this section we try to study the impact of fault on the performance of a real time distributed system. The performance measure considered is throughput.

The task scheduling scheme used for the study is the First-Come-First-Serve (FCFS). Crash fault was injected to upto 10% and 20% respectively. Results as shown in Figure 2.17 and Figure 2.18 clearly reflect that in presence of fault the throughput of the system degrades. Figure 2.19 shows that, unlike the above results, for periodic tasks if a small percentage of processor or nodes, constituting the RTDS, goes faulty then the performance degradation is significant.

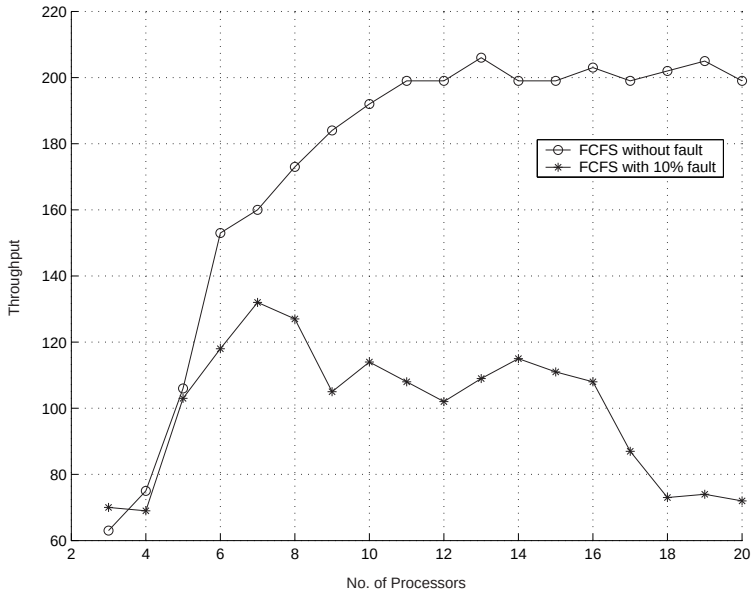


Figure 2.17: Effect of faults upto 10% with FCFS

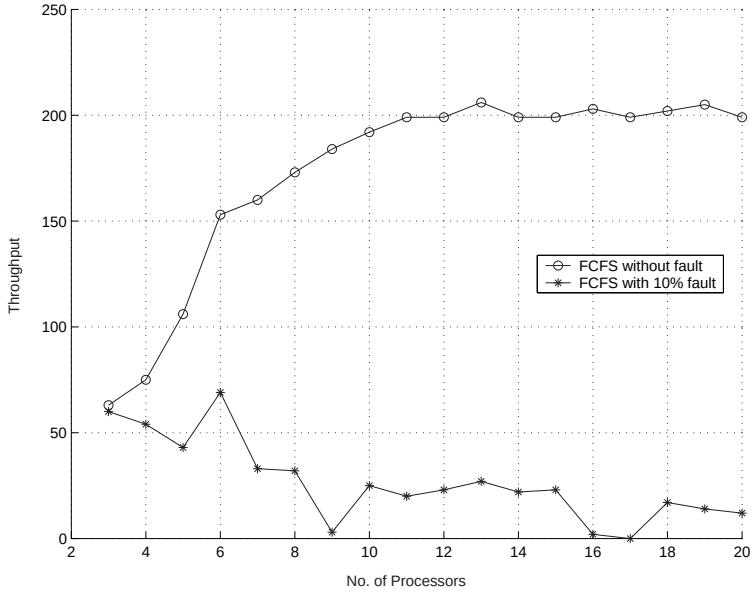


Figure 2.18: Effect of faults upto 20% with FCFS

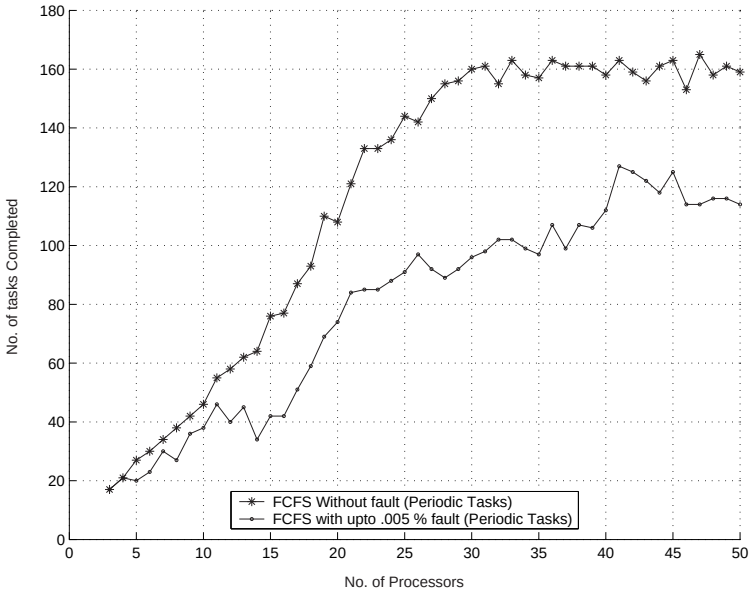


Figure 2.19: Effect of faults upto 1% with FCFS for periodic tasks

2.13 Conclusion

This chapter introduced the basic concepts, the terminology and the state of the art of fault tolerance in distributed real time systems. To this end, the basic concepts of the fault tolerance theory were overviewed, focusing primarily on the hard real-time scheduling of tasks on processors. Model for Real Time Distributed System (RTDS) has been presented, it includes the system architecture and workload model. Study has been done to determine whether there is any impact of faults in a system. The results clearly indicate that in the presence of fault the system performance degrades to a large extent and hence it is imperative that both distributed applications and system support mechanism for fault tolerance to ensure that large scale environment are usable. A brief overview of the fault occurring in RTDS and methods to overcome it have been described.

Chapter 3

Task Scheduling in RTDS

3.1 Introduction

The purpose of real time computing is to execute, by the appropriate deadline. The correctness of computation is based not only be logical correctness but also on time at which the results are produced [101]. The scheduling problem in real time distributed systems can be conceptually separated into two parts. As there are many nodes where a task can be executed, the first question to be answered is how to assign the tasks to them. This assignment is known as task allocation, or global scheduling. Once tasks have been allocated, the problem becomes one of defining a feasible local schedule for each node. The local scheduling problem is equivalent to the scheduling problem in uniprocessor systems. Real time task scheduling essentially refers to determining the order in which the various tasks are to be taken up for execution by operating systems. Scheduling of real-time tasks is very different from general scheduling. Ordinary scheduling algorithms attempt to ensure fairness among tasks, minimum progress for any individual task, and prevention of starvation and deadlock. To the scheduler of real-time tasks, these goals are often superficial. The primary concern in scheduling real-time tasks is deadline compliance. In this chapter we look at techniques for allocating and scheduling tasks on processors to ensure that deadline are met.

Real time applications in an industrialized technological infrastructure such as modern telecommunication systems, factories, defense systems, aircraft, airports and space stations pose relatively rigid requirements on their performance. These requirements are usually stated as constraints on response time and/or on the temporal validity of sensory data. Hence a real-time system requires a high degree of schedulability. Schedulability is the degree of resource utilization at or below which the timing requirements of tasks can be ensured. Many practical instances of scheduling algorithms have been found to be NP-complete, i.e. it is believed that there is no optimal polynomial time algorithm for them [29].

The problem that we consider is how to distribute (or schedule) processes among

processing elements to achieve performance goal(s), such as minimizing execution time, minimizing communication delays, and/or maximizing resource utilization. In a system consisting of real-time transactions, each of which requires computational, communication and data resources to processed, scheduling is the problem of allocating resources to satisfy the requirements of those transactions.

Fault tolerance can be achieved by scheduling multiple versions of tasks on different processors. In general four different model have been proposed by various researchers, which are mostly used for fault tolerant scheduling in real time system. These techniques are (i) Tripile modular redundancy, (ii) Primary Backup(PB) model, (iii)Imprecise computational(IC) model and (iv) (M,k) firm deadline model[30]. In thesis, we have used the basic of primary backup(PB) model to design new fault tolerant scheduling schemes

3.2 Concepts and Terms

Real-time distributed computing involves two aspects related to timeliness: the objective and the means to that objective. In this section a few important concepts and terminologies which would be useful in understanding the rest of this chapter has been stated.

3.2.1 Scheduling and Dispatching

Scheduling is the creation of a schedule: a (partially) ordered list specifying how contending accesses to one or more sequentially reusable resources will be granted. Such resources may be hardware such as processors, communication paths, storage devices or they may be software, such as locks and data objects. A schedule is intended to be optimal with respect to some criteria (such as timeliness ones).

In contrast, dispatching is the process of granting access to the currently most eligible contending entity. Eligibility is manifest either by the entity's position in a schedule (the first entity in the schedule has the highest eligibility of all entities in the schedule) or, in the absence of a schedule, by the value of one or more eligibility parameters, such as priority or deadline (the most eligible one has either the highest priority or the earliest deadline, respectively, of all entities ready to access the resource).

3.2.2 Schedulable and Non-Schedulable Entities

A computing system usually has a mixture of schedulable and non-schedulable entities. Schedulable entities (e.g., threads, tasks, and processes in both the application and the system software) are scheduled by the scheduler (which may be part of some system software, such as an operating system, or an off-line person or program). Non-schedulable entities are most often in the system software and can include interrupt

handlers, operating system commands, packet-level network communication services, and the operating system's scheduler. Non-schedulable entities can execute continuously, periodically, or in response to events; their timeliness is a system design and implementation responsibility.

Real-time computing practitioners (i.e., real-time OS vendors and users) focus primarily on timeliness in terms of non-schedulable entities (e.g., interrupt response times). Real-time computing principles and theory are focused primarily on timeliness in terms of schedulable entities (e.g., meeting task completion deadlines).

At this point it would be useful to define a few very important terms as well:

Definition 3 (Critical Task). *A task is said to be critical if the consequences of not meeting the deadline leads the system to a fatal fault, it can be catastrophic. Periodic tasks usually have deadlines, which belong to this category.*

Definition 4 (Valid Schedule). *A valid schedule of a set of tasks $\forall$ is a schedule of $\forall$ satisfying the following properties:*

- *Each process can only start execution after its release time.*
- *All the precedence and resource usage constraints are satisfied.*
- *The total amount of processor time assigned to each task is equal to its maximum or actual execution time.*

Definition 5 (Feasible schedule). *A feasible schedule of a set of tasks ξ is a valid schedule by which every task completes by its deadline a set of tasks is schedulable according to a scheduling algorithm if the scheduler always produces a feasible schedule.*

Definition 6 (Optimal schedule). *An optimal schedule of a set of tasks ξ is valid schedule of ξ with minimal lateness. A hard real-time scheduling algorithm is optimal if the algorithm always produces a feasible schedule for a given set of tasks.*

Definition 7 (Efficient scheduling). *Efficient task schedule is the one that minimizes the total completion time, or the schedule length, of the application.*

3.2.3 Timeliness Specification

In real time systems the time is qualified which determines the type of real time system. The timeliness specification can be of the following type.

3.2.3.1 Deadline

A deadline is a completion time constraint which specifies that the timeliness of the task's transit through the deadline scope depends on whether the task's execution point reaches the end of the scope before the deadline time has occurred, in which case the deadline is satisfied.

3.2.3.2 Hard Deadline

A hard deadline is a completion time constraint, such that if the deadline is satisfied i.e., the task's execution point reaches the end of the deadline scope before the deadline time occurs, then the time constrained portion of the task's execution is timely; otherwise, that portion is not timely.

3.2.3.3 Soft Deadline

A soft deadline is a completion time constraint, such that if the deadline is satisfied i.e., the task's execution point reaches the end of the deadline scope before the deadline time occurs then the time constrained portion of the task's execution is more timely; otherwise, that portion is less timely. Thus, a hard deadline is a special case of a soft deadline.

3.2.4 Hard Real-time

Hard real-time is the case where: for the schedulable entities, some time constraints are hard deadlines, and the timeliness component of the scheduling optimization criterion is to always meet all hard deadlines (additional components may apply to any soft time constraints); for the non-schedulable entities, some upper bounds are hard, and the system has been designed and implemented so that all hard upper bounds are always satisfied (other non-schedulable entities may have soft upper bounds). Thus, the feasible schedules (with respect to those schedulable entity time constraints) are always optimal, and the predictability of that optimality is maximum (deterministic). Hard real time systems can be hence defined as

Definition 8 (Hard Real Time System). *Hard real time systems are based on deadline schemes, usually using priority as well. Such systems typically have a* worst case *requirement. Failure to meet timing requirement leads to fatal fault and failure to meet a deadline, in such systems, requires automated handling,*

3.2.5 Soft Real-time

Soft real-time represents all cases which are not hard real-time (soft real-time is the general case, of which hard real-time is a special case). Time constraints are soft (which may include the hard deadline special case), such as the classical lateness function. Any scheduling optimization criteria may be used (including the hard real-time special case), such as minimizing the number of missed deadlines, or minimizing mean tardiness, or maximizing the accrued utility. Predictability of schedule optimality (and thus thread timeliness) is generally sub-optimal, but may be deterministic (including but not limited to the special hard real-time case). Upper bounds are soft, and predictability of non-schedulable entity timeliness is generally sub-optimal. Soft real time systems can be hence defined as

Definition 9 (Soft Real Time System). *Soft real time systems have an average case timing requirement. Failure to meet the timing requirement is not critical in such systems. Such systems are often based on priority schemes*

Unlike in hard real time systems where it becomes important to ensure all tasks are completed by their deadline in soft real-time systems where meeting the deadlines of all the tasks is not essential.

3.3 Taxonomy of Real Time Scheduling Algorithms

The vast majority of scheduling/assignment problems on systems with more than two processors are NP complete [28]. The problem of scheduling in multiprocessor and distributed systems is reduced to that of uniprocessor scheduling [62]. In general, tasks may have data and control dependencies and may share resources on different processors.

At the highest level, a distinction is drawn between hard and soft scheduling, depending on the timing constraint defined as hard and soft real time systems.

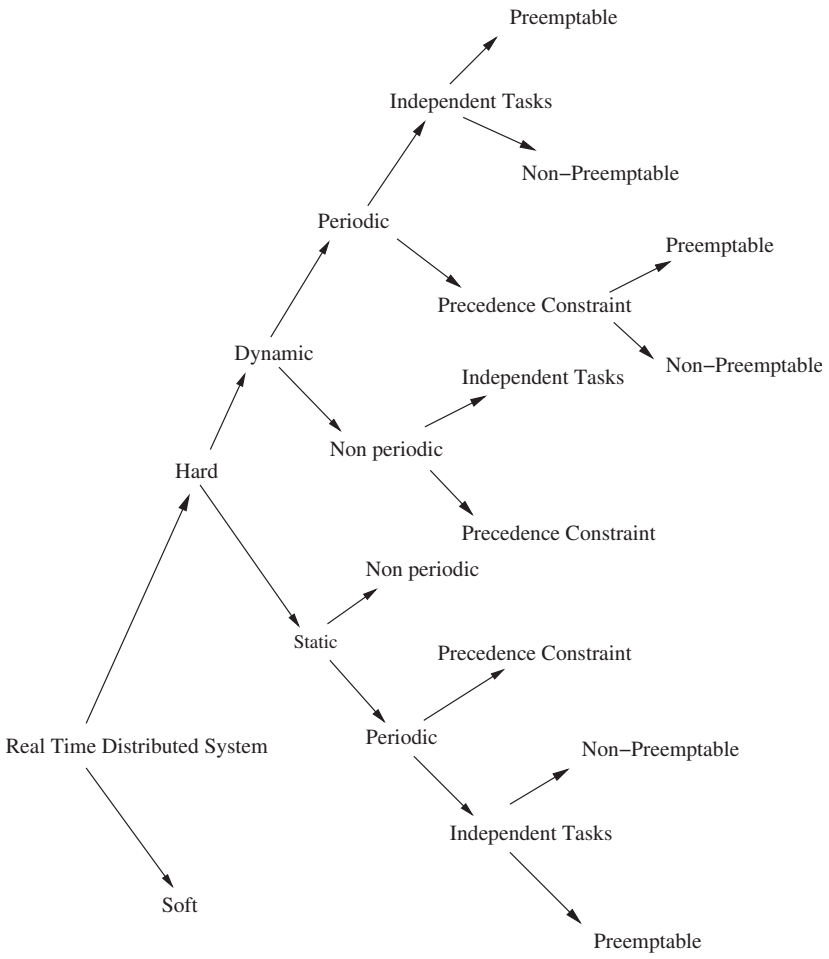


Figure 3.1: Taxonomy of Real Time Scheduling

The requirements of scheduling approaches in distributed, critical, real time systems includes comprehensive handling of preemptable and non-preemptable tasks, periodic and non-periodic tasks, multiple importance level of tasks, groups of tasks with a single deadline, end-to-end timing constraints, precedence constraints, communication requirements, resource requirements, placement constraints, fault tolerance needs, tight and loose deadline, and normal and overload conditions. In this list of requirements the key issues is to deal with task set and environment in multiprocessing and distributed systems and emphasize not only on the accuracy but on predictability also [102].

In a simple system, the ability to demonstrate at design time that the constraints of all tasks can be met with 100% certainty is called predictability. We cannot predict which task will meet all its constraints at design time. However, each tasks knows whether its constraints can be satisfied while the system is in operation. For some complex systems, the semantics of predictability varies from one task to another. Some critical tasks may still require a 100% guarantee that their constraints will be satisfied. Periodic tasks with deadlines usually belong to this category [98].

The researchers have proposed several task scheduling schemes for processor allocation among multiple computing entities. These different algorithms have been classified to get a better understanding of the issues involved, and for comparison. These schedulers dynamically determine the feasibility of scheduling new tasks arriving to the system. An abstract model of scheduler is depicted in Figure 3.2.

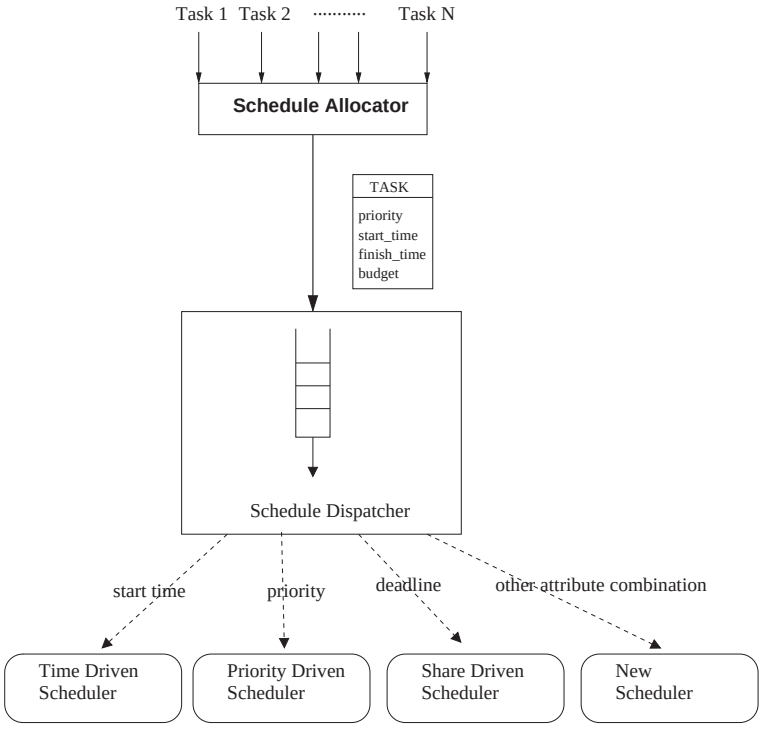


Figure 3.2: Generalized Scheduling Framework in RTDS

3.4 Schedulability analysis

To guarantee that the application timing constraints will always be met, one must do a pre-runtime schedulability analysis of the system. From the scheduling point of view, distributed real time systems impose much more problem and constraints than centralized systems. In distributed systems the problem is not only to decide "when" to execute a task, but also to decide "where" the task should be executed. It is up to the global scheduler to decide which is the most suitable mode to execute a given task and to give a warranty that it will be able to schedule all the tasks and network traffic, meeting all the application-timing constraints. The scheduling analysis consists of two

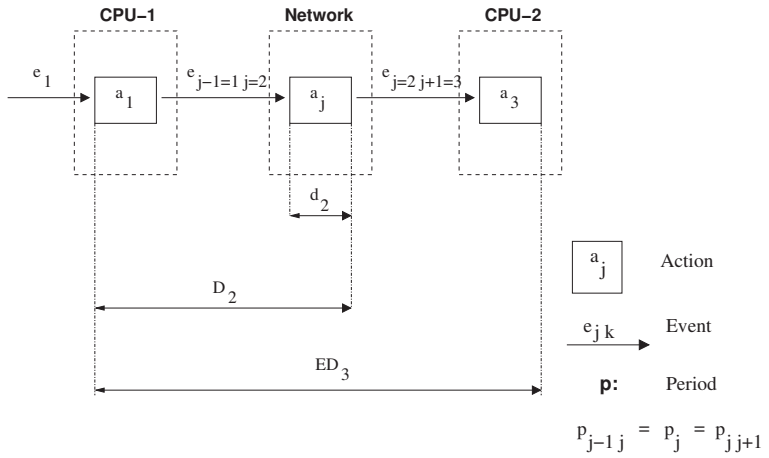


Figure 3.3: Linear model for event driven distributed system

parts based on the linear model shown in Figure 3.3.

1. For each task its worst-case response time is calculated. The response time is the time needed by a task to be completed, including preemption due to higher priority tasks.

2. The same is done for message transmissions.

Based on this response time, denoted by R of all task chains are calculated and it is tested whether one or more deadlines are missed. The scheduling analysis for priority ordered execution in its simplest form can be expressed by the following fixed-point equation.

$$R_i^{n+1} = c_i + \sum_{j \,\epsilon\, hp(i)} \left\lceil \frac{R_i^n}{t_j} \right\rceil c_j \tag{3.1}$$

where $hp(i)$ is the set of tasks running on the same node with higher priority. The iteration either ends with some n when $R_i^{n+1} = R_i^n$ or when R_i^n exceeds the given deadline.

There are some fundamental differences message scheduling and task scheduling. Although messages are somewhat different from tasks, a lot of analogies do exists

between task scheduling and message scheduling. They have many points in common, as shown by the following :

1. Scheduling tasks on processor means to serialize their execution on a processor. Similarly, it is possible to consider message scheduling through a network as serializing the message transmission through the physical medium.

2. Two tasks cannot be executed on the same processor at the same time, as well as two messages cannot be transmitted through a physical support at the same time (otherwise collision will occur)

3. A task has computation time depending on the number of executed instructions. Similarly a message has transmission time depending on the number of bits to transmit.

An appropriate way to model message transmission is to exploit analogy between arbitration of the communication medium by message and CPU arbitration on the node. In order to model message transmission, each message is assigned a unique priority. Messages to be sent are stored in a priority-ordered queue. Using priority driven bus, the time needed for transmission of a message m_i, now can be calculated by

$$R_{m_i}^{n+1} = \rho n_i + I_{m_i} \; with \; I_{m_i} = \sum_{m_j \, \epsilon \, hp(m_i)} \left\lceil \frac{R_{m_i}^{n+1}}{t_{m_j}} \right\rceil \rho m_j \tag{3.2}$$

The task of developing a multiple processor scheduling is therefore divided into three steps: first we assign tasks to processors, and second we, run uniprocessor scheduling, if the schedule is not feasible or the tasks cannot be completed within their deadline then reallocation of the tasks takes place this process is summarized in Figure 3.4.

3.5 Task Assignment

The optimal assignment of tasks to processors is, in almost practical cases, an NP-complete problem. Hence applying heuristic techniques. These heuristic cannot guarantee that an allocation will be found that permits all tasks to be feasibly scheduled. All that can be done is to make an allocation, check its feasibility , if the allocation is not feasible, modify the allocation to render that the allocation is feasible. Sometimes the allocation uses the communication cost as the part of the allocation scheme.

Task allocation in distributed systems is a combinatorial optimization problem. The set of all possible assignments of tasks to processors defines the solution space W. An allocation $\mho \, \epsilon \, W$ is known as a solution of the problem. In order to represent the quality of each allocation, a function $E(\mho) : W \rightarrow \Pi$ is defined that quantifies, for each $\mho \, \epsilon \, W$, how well it satisfies the optimization goals. The greater the value returned by $E(\mho)$, the farther is the solution from the desired optimum. This function

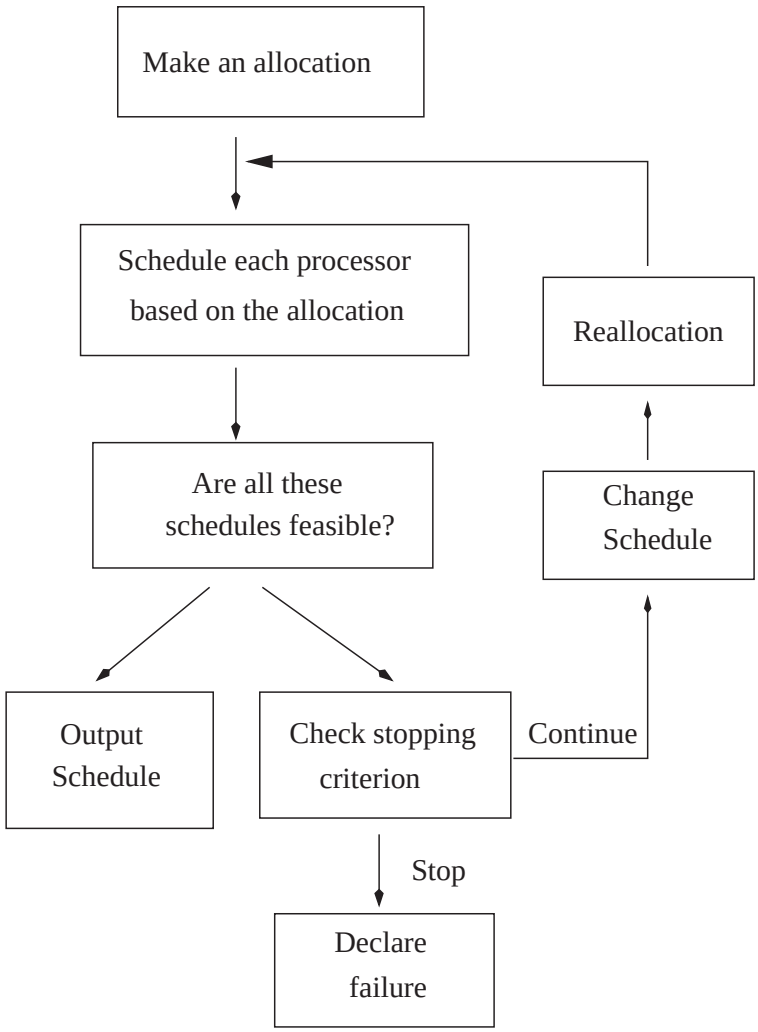


Figure 3.4: Developing a feasible real time distributed system schedule

is normally named cost function or sometimes energy function. As an example of the role of such function, consider the case of two tasks that perform exactly the same job. The tasks are replicas intended to execute in different nodes of a distributed system for fault-tolerance reasons. If a solution has the two tasks assigned to the same node, an objective of the system designer is clearly violated. Therefore, the cost of that solution must assume a greater value than that of other solutions in conformity with the fault-tolerance constraint, if other parameters are maintained the same.

In order for a heuristic to be able to explore the solution space of a problem, it is necessary to define, given an arbitrary allocation $\mho$ being evaluated at the moment, which are the possible candidates for evaluation in the next step. The set of such solutions is named the neighborhood set of $\mho$, and is noted by $nbd(\mho)$. An allocation $\mho* \in nbd(\mho)$ is known as a neighbor of $\mho$. We will allow two kinds of operations (also named moves) through which $\mho$ can be transformed in a neighbor $\mho*$:

1. Simple move: a task assigned to node P_i is moved to node P_j, where $i \neq j$;

2. Double move: a task assigned to node P_i changes position with a task assigned

to node P_j, where $i \neq j$.

Only one of these moves can be applied, at each step of the heuristic, to transform $\mho$ into a neighbor $\mho*$. The set of movements described defines $nbd(\mho)$.

Two more concepts related to the cost function are essential for the discussion in the following section. A local minimum is a solution $\mho \epsilon W$ such that $\forall \mho * \epsilon nbd(\mho)$, $E(\mho) \leq E(\mho*)$. A global minimum correspond to a solution $\mho \epsilon W$such that $\forall \mho \epsilon W$, $E(\mho) \leq E(\mho)$. Clearly, what is desired through the application of any heuristic is to achieve the global minimum. Table 3.1 summarizes the task assignment schemes.

3.6 Task Scheduling

This section presents an historical review of scheduling literature. Of course, this is not an exhaustive list of all the literature; it encompasses what we believe to be important studies. An adaptive resource management technique is discussed in [83].

Three commonly used approaches to Uniprocessor scheduling real time systems: (i) clock driven, (ii) weighted round robin and (iii) priority-driven (iv) Cyclic Scheduler.

3.6.1 Clock Driven Approach

In this approach the decision on what job executes at what time are made at specific time instants. These time instants are chosen before the system begins execution. System that uses clock driven scheduling, all the parameters are fixed and known A schedule of the job is computed off-line and is stored for use at run time. The scheduling overhead during run time can be minimized.One way to implement such a schedule is the use of a timer.

3.6.2 Weighted Round Robin Approach

Time shared applications use this approach. Every job joins a First-in-first-out (FIFO) queue when it becomes ready for execution. The job at the head of the queue executes for at most one time slice. The weighted round-robin algorithm has been used for scheduling real time traffic in high speed switched network. The completion time of the job is delayed since it uses a fraction of the processor and hence the response time is unduly large. For this reason, the weighted round robin approach is not suitable for scheduling critical jobs.

3.6.3 Priority Driven Approach

Priority driven approaches are event driven. Jobs ready for execution are placed in one or more queues by the priorities of the jobs. At any scheduling decision time, the jobs with the highest priority are scheduled and executed on the available processors.

Author	Ref	Scheme	Remark
Bannister Trivedi	[10]	Utilization Balancing algorithm	The goal of the algorithm is to balance the task utilization.
Davari Dhall	[23]	Next-Fit algorithm	Uses task set RM Scheduling. Assumes that it consists of identical processors.
Coffman	[19]	Bin-Packing algorithm	Uses task set of EDF. Many algorithm exists to solve the problem e.g. First-Fit and next fit algorithms and their extensions, such as RMFF [24], RMST and RMGT [14] algorithms, are good solutions.
Ramamritham Shiah Stankovic	[84]	Myopic offline scheduling	For non-preemptive tasks. MOS proceeds by building up a schedule tree.
Ramamritham Zhao Stankovic	[86]	Focussed addressing and bidding algorithm	An offline procedure. Is used for task sets consisting of both critical and non-critical real time tasks.
Shin Chang	[97]	Buddy strategy	Similar to FAB. Processor selection strategy is different
Shin Krishna	[53]	Assign set of task processors according to their LFT	Algorithm is trial and error task to process.
Prayati Koulamas Koubias Papadopoulos	[78]	FBALL algorithm	Hybrid approach. Priorities according to Priority slicing technique.
Abdelzaher Shin Peng	[76]		Uses a new B&B algorithm, uses a new bounding function.
Maheswaran Ali Siegel Hensgen	[70]	Max-Min Min-Min Sufferage	Uses a batch mode algorithm and immediate mode algorithm

Table 3.1: Task Assignment Schemes on Distributed Real Time Systems.

The priority list and other rules, such as whether preemption is allowed, define the scheduling algorithm completely. In general non-preemptive scheduling is not better than preemptive scheduling. Its difficult to decide when to schedule jobs preemptively or non-preemptively. In a multiprocessor system, the minimum makespan achievable by an optimal preemptive algorithm is shorter than that of optimal non-preemptive algorithm. When there are two processors, the minimum makespan achievable by non-preemptive algorithm is never more than 4/3 times the minimum makespan achievable by preemptive algorithm when cost of preemption is negligible [20]. A priority driven system is nondeterministic when job parameters vary.

The algorithms for scheduling periodic tasks are of two types: fixed priority and dynamic priority. A fixed-priority algorithm assigns the same priority to all the jobs in each task. In contrast, a dynamic-priority algorithm assigns different priorities to the individual jobs in each task. Hence the priority of the task with respect to that of the other tasks changes as jobs are released and completed.

3.6.4 Cyclic Scheduler

To use this type of scheduler, the number of periodic tasks and the parameters of the tasks in the system at all times must be known a priori. The scheduler is computed offline based on this information and stored in as a table for use bye the scheduler at run time. According to such schedule, scheduling decisions are made at the beginning of each frame, the scheduling decision time partition the time line into intervals called frames, there is no preemption within each frame.At the beginning of each frame the scheduler verifies that whether all the jobs are completed within their deadline. The scheduler takes appropriate recovery action if any of the conditions are not satisfied.

A review of literature on Uniprocessor scheduling of periodic tasks is shown in Table: 3.2. The real time scheduling has been also described in [85]. Global scheduling techniques in Distributed Real Time Systems are discussed in [89]. [91] Minimizes the completion time of parallel programs by distributing cooperating concurrent tasks to homogeneous networked nodes (NOW).

3.7 Comparison of different assignment algorithms

The applicability and strength of heterogeneous computing systems are derived from their ability to match computing nodes to appropriate tasks since a suite of different machines are interconnected. A good mapping algorithm offers minimal expected completion time and machine idle time. We do a comparative study to find the optimal dynamic task assignment algorithm of independent task, by measuring the efficiency of each algorithm, with different arrival patterns. Efficiency of the algorithm is determined by the ratio of the number of task completed to the number of task arriving to the system in a fixed interval of time. We have considered two batch

Author	Ref.	Sched Type	Inter. type	Model Input	Task Exec	Deadline assign.	Priority assign.
Liu Layland	[61]	S	P	UT	K	=Period	Period
Liu Layland	[61]	D	P	UT	K	=Period	Deadline
Abdelzaher Shin	[2]	S	P	PG	K	relative to arrival time	Deadline
Jeffay Stanat Martel	[44]	S	NP	UT	K	=Period	Period
Natale Stankovic	[71]	D	P	UT	K	=Period	Deadline
Liu	[63]	S	P	UT	K	=release time	-
Liu	[63]	-	P	-	K	arbitrary	Slacks
Jawad	[43]	D	P	-	K	-	-
Mok Wang	[71]	D	P	-	K	=ready time of next job	-
Ramamurthy Moir	[88]	S	P	-	K	pseudodeadline related to period	-
Ramamritham Fohler	[26]	S	P	PG	K	≤ period	-
Chung Liu Lin	[113]	S	P	-	K	= period	-

Table 3.2: Comparison of models found in the Uniprocessor scheduling of periodic tasks literature. The type of scheduling can be either Static, Dynamic or Hybrid (static then dynamic). The interruption type can be Preemptive or Non-preemptive. The model inputs can be an Arbitrary Graph, a Tree, a Precedence Graph, a DAG, or Unrelated Tasks. Task execution time is either Known or Unknown.

mode, Max-Min and Min-Min [67] and two immediate scheduler, Earliest First (EF) and Lightest Loaded (LL) [1].

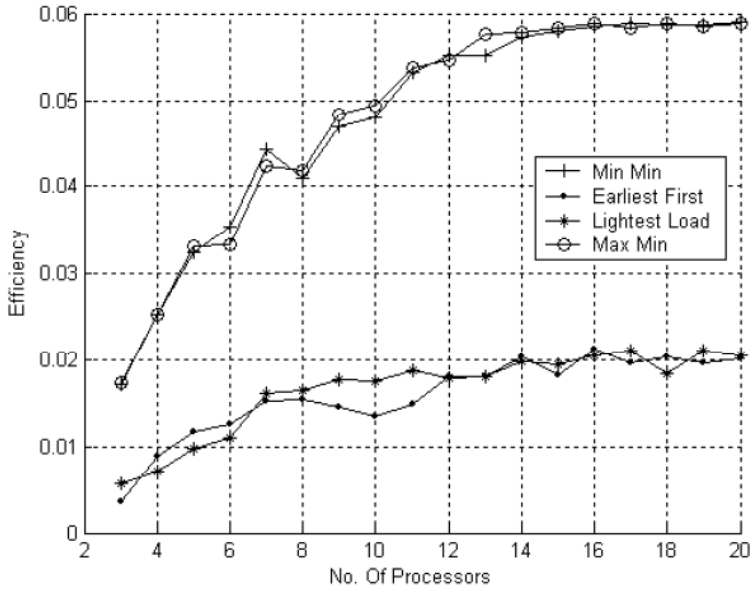


Figure 3.5: Performance with task arrival over Poisson distribution

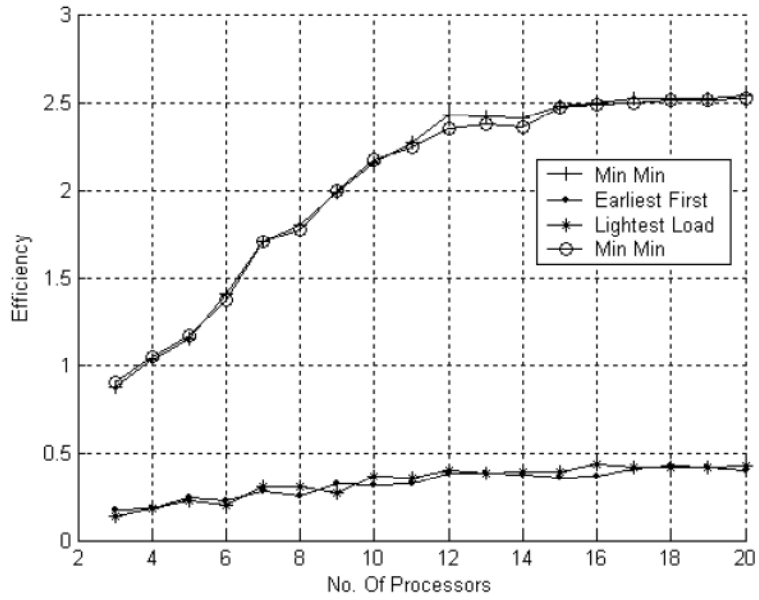


Figure 3.6: Performance with task arrival over Uniform distribution

The results, Figure 3.5, 3.6 and 3.7 clearly shows that the batch mode tasks assignment schemes perform better that the immediate tasks assignments algorithms. The uniprocessor scheduling scheme is FCFS on the processor to which the tasks are mapped or assigned. A comparison among the two batch mode task assignment algorithms shows that Max-Min has a lower completion time of increasing number of tasks as shown in Figure 3.7. The comparisons were made upon tasks on a typical

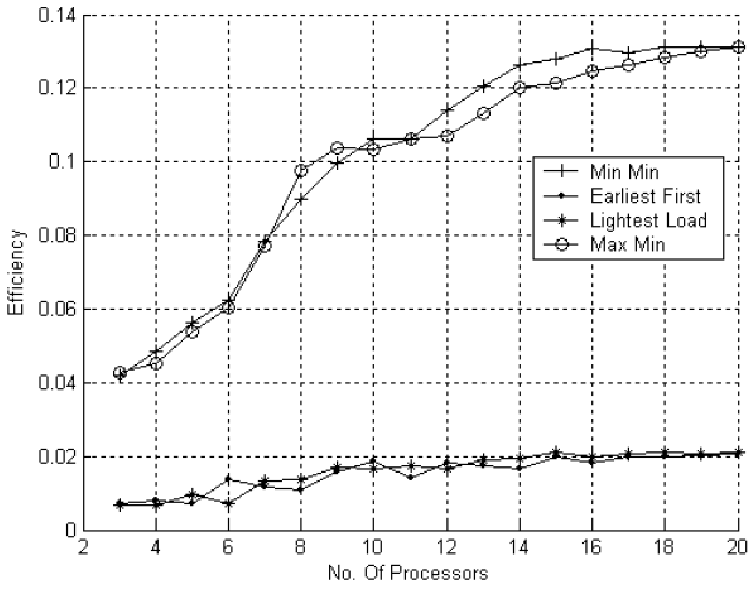


Figure 3.7: Performance with task arrival over Normal distribution

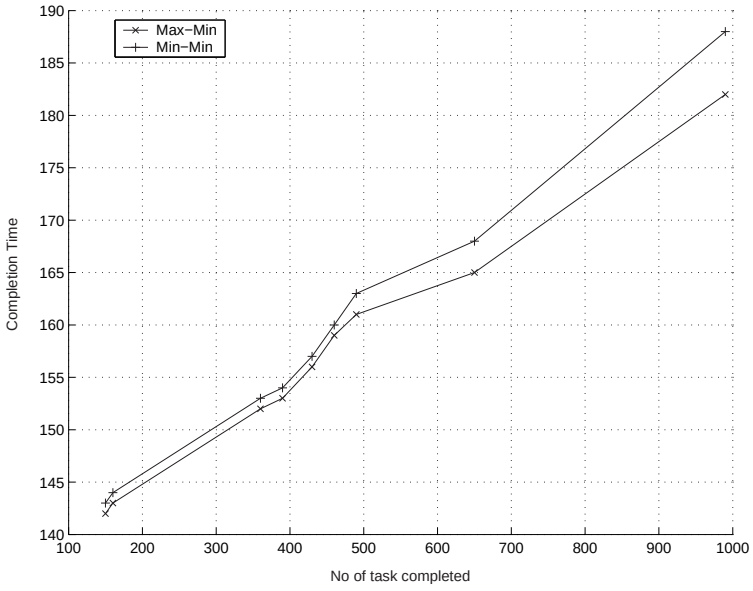


Figure 3.8: Comparison between Min-Min and Max-Min with task pool

heterogeneous distributed system, prominently identified by worst case execution time (WCET). We then took up the study to find the optimal task assignment for periodic tasks which are the most appearing task in real time system distributed system. We considered the Max-Min, First-Come-First-Serve [106] and Randomized task allocation algorithm [64] with Poisson arrival distribution. Figure 3.7 depicts the performance of the three task assignments algorithms. The performance of Max-Min turnouts to be poor as compared to others.

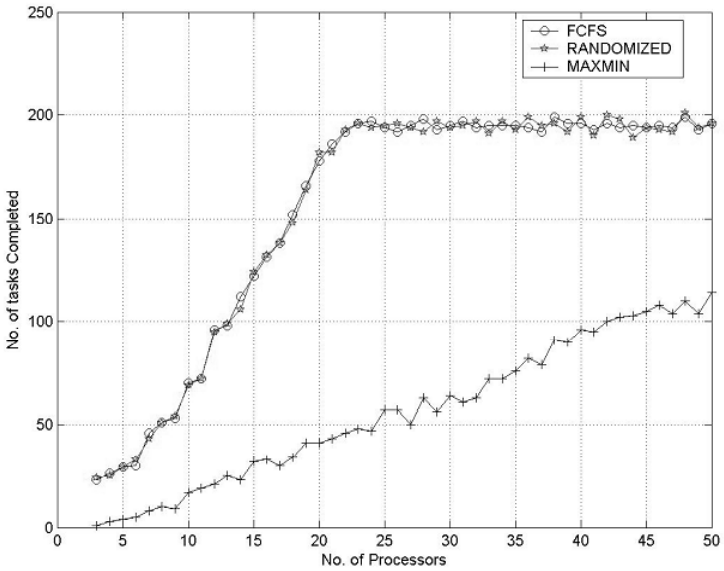


Figure 3.9: Performance comparison of Max-Min, FCFS and Randomized for periodic tasks.

3.8 Conclusion

In this chapter we have studied the task scheduling methodology and reviewed scheduling algorithms in real time distributed systems. Task scheduling in real time distributed system is performed in two broad steps. First, the task assignment or mapping of tasks to the processors or nodes in the system. Second is the uniprocessor scheduling. A batch mode vs immediate mode comparisons of dynamic task assignment schemes has been done. Batch mode task assignment scheme performance is good than for task, which are prominently identified by its WCET. But on further investigation of a batch mode scheduler with FCFS and Randomized task assignment algorithm for periodic tasks occurring in real time system. FCFS and Randomized performed better than Min-Min task assignment scheme. All the comparisons were done for efficiency. We have hence, used the Randomized approach for task assignment in our system for achieving fault tolerance, as described in the upcoming chapter. We have selected preemptive EDF as uniprocessor scheduler for fault tolerance, since it is optimal for uniprocessors.

Chapter 4

Fault Tolerance Techniques in RTDS

4.1 Introduction

Computational tasks continue to become more complex and require increasing amounts of processing time. At the same time, high performance computer systems are composed of increasing numbers of failure prone components. The end result is that long running, distributed applications are interrupted by hardware failures with increasing frequency. Additionally, when an application does fail, the cost is more higher since more computation is lost. It is imperative that both distributed applications and parallel system support mechanism for fault tolerance to ensure that large scale environment are usable.

A "fault-tolerant system" is one that continues to perform at desired level of service inspite of failures in some components that constitute the system. Research on fault-tolerant real time system design has always been around from NASA's deep space probe to Medical Computing devices (e.g. pacemakers). It becomes more relevant with the real world problems increasing dependency on such automated critical system on which a single failure may have a catastrophic effect on system performance. Extreme fault tolerance is required in car controllers (e.g. anti-lock brakes). Commercial servers (databases, web servers), file servers, etc. require high fault tolerance. Applications running on Desktops, Laptops, PDAs, etc. are also require some fault tolerance. Hence, there is a need for efficient fault tolerant techniques, which provides guarantee to the real time tasks in execution, and thereby providing reliability and dependability to the system.

Faults in a Real Time Distributed System (RTDS) may appear either in the hardware or in the software and they can be classified as being permanent, intermittent or transient [38, 22, 36]. One major advantage of distributed systems is to tolerate individual component failure without terminating the entire computation [47, 53]. Research in fault-tolerant real time distributed computing, aims at making real time distributed systems more reliable by handling faults in complex computing environments. More-

over, the increasing dependence of different services on real time heterogeneous distributed system has led to an increasing demand for dependable systems, systems with quantifiable reliability properties. Faults in distributed can arranged in hierarchy as mentioned in section 2.7.

In Fault Tolerant Real Time Distributed Systems (FTRTDS), detection of fault and its recovery should be executed in timely manner so that in spite of fault occurrences the intended output of real-time computations always take place on time. For fault-tolerant technique fault detection, latency and recovery time are important performance metrics because they contribute to system's downtime. A fault tolerant technique can be useful, in Real Time Distributed System if its fault detection latency and recovery time are tightly bounded. When this is not feasible, the system must attempt fault tolerance actions that lead to the least damages to the applications mission and the systems users.

Hardware and software redundancy are well-known effective methods for fault-tolerance, where redundant hardware (e.g., processors, communication links) and redundant software (e.g., tasks, messages) are added into the system to deal with different type of faults [53]. However, hardware fault-tolerant techniques are not preferred in most systems due to the limited resources available, extra weights, encumbrance, energy consumption, or price constraints.

The performance of FTRDS is depends on the number of tasks are completed within their specified deadline in presence of faults in the system. The issue of scheduling on heterogeneous systems has been studied in the literature in recent years. Fault tolerance has typically been approached from a hardware standpoint, with multiple replicas of essential components running applications mostly in parallel fashion. Fault-tolerant scheduling strategies in real time distributed systems are described in [116, 82, 83, 114]. The basic requirements of a fault tolerant scheduling algorithm in real time distributed systems are described in [50, 38].

In this chapter, we describe the proposed distributed recovery block based fault tolerant scheduling algorithm for periodic real time tasks. Our algorithms handles permanent, transient and timing faults and ensures that updating the backup task according to primary task works better instead of executing the backup task in parallel.

4.2 Proposed Model

The methodologies described in this section focuses on providing fault tolerance for applications running on distributed systems in real time environment. In these environment, there are many opportunities for failures which affect the entire system as scheduling and synchronized data is lost. This includes, but are not limited to, the processor, disk, memory or network interface on the node.

We consider the basic Distributed Recovery Block (DRB) concepts by Kim and Welch to provide fault tolerance in RTDS, described in section 2.10.4.2. We have used DRB to deal with permanent, transient and timing faults. The underlying design philosophy

behind the DRB scheme is that a real-time distributed or parallel computer system can take the desirable modular form of an interconnection of computing stations, where a computing station refers to a processing node (hardware and software) dedicated to the execution of one or a few application tasks [100]. The idea of the distributed recovery block (DRB) has been adapted from [50, 46]. Recovery block consists of one or more routines, called try blocks here, designed to compute the same or similar result, and an acceptance test which is an expression of the criterion for which the result can be accepted both in term of correctness and timing constraint. For the sake of simplicity a recovery block consists of only two try blocks, i.e. primary and backup [109, 34]. The error processing technique used is acceptance test that is parallel between node pairs but sequential in each node.

In this work we have outlined the two requirements for DRB variant i.e. Primary-Backup fault tolerant algorithm in RTDS

1. the execution of backup versions should not hinder the execution of the primary version of the tasks, and

2. when the primary task fails to meet its deadline the backup instance should then be executed but it should be executed from the point of last correct sub task executed by the primary version.

The first requirement is satisfied by not assigning the backup task to the processor for execution though scheduled to the processor the second requirement is satisfied by the primary task communicating with the processor on which the backup task has been scheduled and updates the backup to the last known successful state of itself. The fault tolerant technique makes sure that the backup tasks though scheduled to processors do not hamper the execution of primary tasks at the same time the backup task are updated according to the subtask completed in their primary counterpart so that when the primary task fails the backup task does not start its execution from beginning instead starts from the last updated subtask. When the primary task is completed within its deadline the backup task is terminated. The architecture for achieving fault tolerance in presence of permanent, transient and timing fault in the real time distributed system has been shown in Figure 4.1.

It consists of central queue, CQ at which the tasks arrive. From CQ the tasks are assigned to different processors. Each processor consists of two queues the primary queue, PQ_{P_k} and a backup queue, BQ_{P_k}. Tasks in BQ_{P_k} can be moved to PQ_{P_k}. It should be noted that only the task in PQ_{P_k} can be execute by the processor P_k. The PQ_{P_k} communicates with the back up queue of other processor, BQ_{P_l}, where $k \neq l$.

The tasks with computational requirement arrive from the external real time environment to the central queue following a poisson distribution. The tasks are allocated from the central queue on a FCFS basis using the Random allocation scheme, to the computing nodes or processor constituting the RTDS. Once the task is selected for allocation a copy of the task is generated called as the backup task while the former

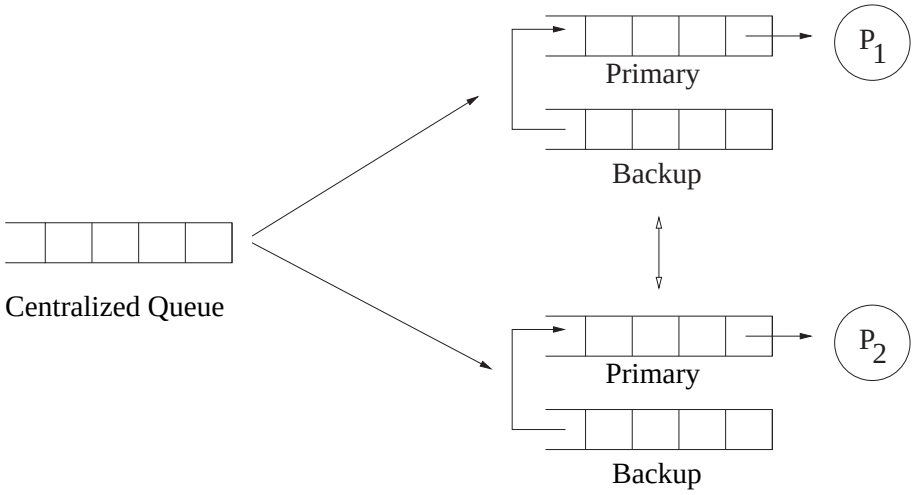


Figure 4.1: Fault Tolerant RTDS architecture for 2 nodes

is called the primary task and represented as $Back_{T_i}$ and Pri_{T_i} respectively. The two versions of the task are allocated to different processors. Each processor executes the tasks in the queue using Earliest Deadline First (EDF) Algorithm [53, 107, 62]. Unlike the state-of-art techniques where both the primary and backup tasks are executed in parallel, in this fault tolerant architecture only the tasks in primary queue of each processor are executed and not the backup tasks.

Checking the intermediate deadline of the primary task when the given percentages of tasks are completed and the processor status monitors the fault in the system as shown in Figure 4.2. If the task misses the intermediate deadline or the processor

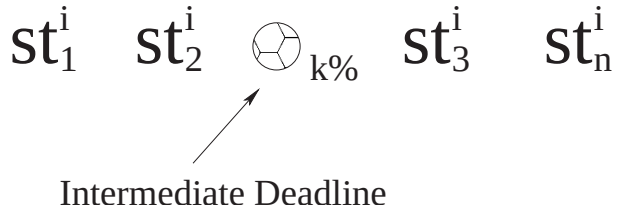


Figure 4.2: Subtasks and Intermediate Deadline

on which the primary task has been assigned crashes, fault alarm is triggered and the backup task is shifted to the primary queue of the same node. The system's communication links are assumed to be fault-free. The techniques have been evaluated for fixed number permanent and arbitrary timing and transient faults. The performance metric used are throughput (TP) and the guarantee ratio (GR) , as given in equation 4.1 and 4.2 respectively, of effective tasks.

$$\text{TP} = \frac{\text{Total Number of Tasks Completed}}{\text{Observed period of time}} \tag{4.1}$$

$$\text{GR} = \frac{\text{Total Number of Tasks Guaranteed}}{\text{Total Number of Effective Tasks}} \tag{4.2}$$

We have proposed three new algorithms for fault tolerance in RTDS notably fault injection, fault tolerant and adaptive fault-tolerant algorithms. Performance of these techniques are studied for both the independent task and precedence constraint tasks, details about the task has been discussed earlier in section 2.3. The basic building blocks of out techniques has been outlined in Figure 4.3. In fault tolerant algorithm

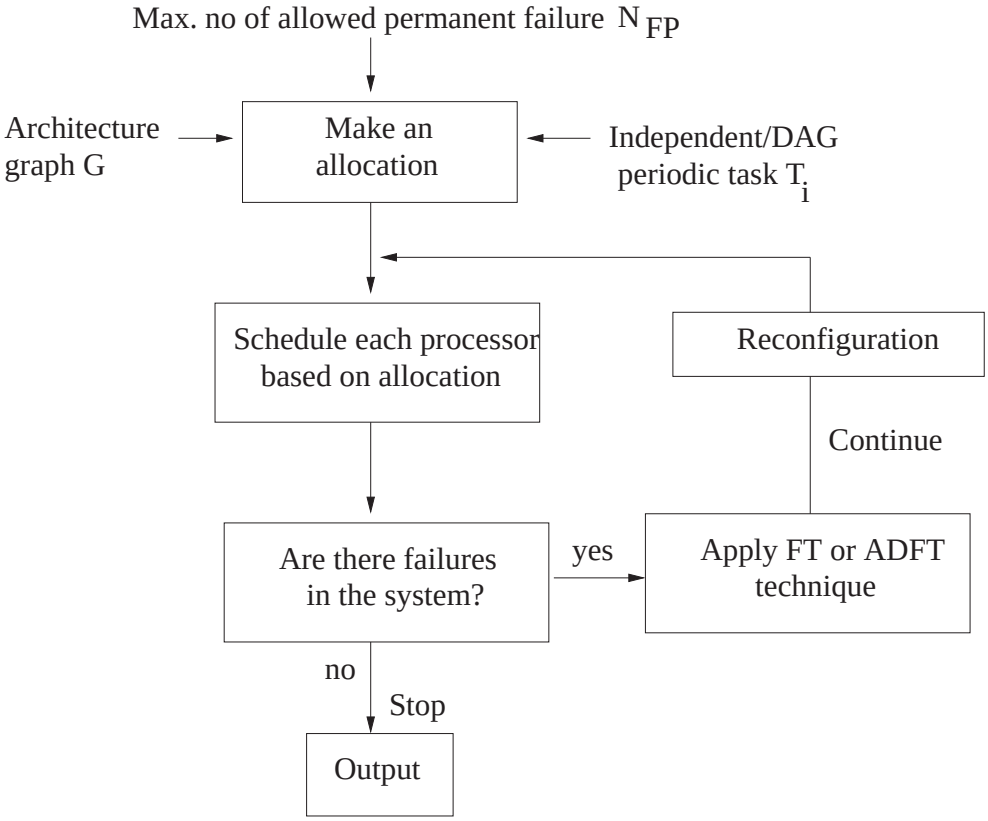


Figure 4.3: Basic building blocks for proposed methodology

Pri_{T_i} and $Back_{T_i}$ are assigned to different processors. If the task misses its deadline or the processor on which the primary task has been assigned crashes, the backup tasks are shifted to primary queue of the same node. The proposed adaptive technique uses the same principle as for fault tolerant technique except that tasks Pri_{T_i} is assigned to the best processor among the two chosen for distribution of primary and backup respectively. $Back_{T_i}$ is assigned to the secondary processor. If the task misses its deadline or the processor on which the primary task has been assigned crashes, the backup tasks are shifted to primary queue of the same node. When the backup task has been added to primary queue an additional replica of the backup task is generated and scheduled to another node in the system. The system's communication channel is assumed to be fault free.

There are three fundamental classes of faults that can occur in a RTDS, also considered in this work. First, is a crash or hang of software on one of many computation nodes the RTDS. In such failures the component can no longer function, but the other component which do not receive any output from the failed component continue to

function.

The second class of failure is a timing fault. Timing fault occurs when the task fails to meet its deadline. This kind of fault have a cascading effect when there are tasks dependent on the failed task. Finally, transient fault, the failed component revives in such faults.

We compare the performance of Adaptive FT Random-EDF, FT Random-EDF with Faulty Random-EDF using a discrete event simulator developed by us using Matlab 6.0.

4.3 Fault Injection Technique

Dependability evaluation involves the study of failures and errors. The destructive nature of a crash and long error latency make it difficult to identify the causes of failures in the operational environment. It is particularly hard to recreate a failure scenario for a large, complex system. Hence, engineers most often use low-cost, simulation based fault injection to evaluate the dependability of a system that is in the conceptual and design phases [108]. Fault injection tests fault detection, fault isolation, and reconfiguration and recovery capabilities [40].

In recent years, researchers have taken more interest in developing software-implemented fault injection tools. Software fault-injection techniques are attractive because they don't require expensive hardware. Furthermore, they can be used to target applications and operating systems, which is difficult to do with hardware fault injection. In this section a fault injection algorithm has been described as mentioned in Algorithm 1.

Algorithm 1 (Fault Injection Algorithm).

1: ***Input:****a system resource set* Π
2: Select a processor P_i
3: ***if*** *No. of Faults* $(P_i) < N_{FP}$ ***then***
4: *Mark* P_i *as FAULTY*
5: *Increment the Upper Limit*
6: ***end if***

In this technique, a timer expires at a predetermined time, triggering injection. Specifically, the time-out event generates an interrupt to invoke fault injection. The timer can be a hardware or software timer. This method requires no modification to the application or workload program. A hardware timer must be linked to the system's interrupt handler vector. Since it injects faults on the basis of time rather than specific events or system state, it produces unpredictable fault effects and program behavior. It is, however, suitable for emulating transient faults and intermittent hardware faults.

When the timer is set we set the flag associated with the randomly selected processor, to indicate that the fault has occurred. The timeout mechanism continues till the time for which the system is observed or the maximum allowed faults, whichever is earlier. The maximum number of faults that can occur is 10% and 20%, of the total system, respectively. The given fault injection technique has been used both in fault tolerant and adaptive fault tolerant technique.

4.4 Fault Tolerant Technique

The proposed algorithm is based upon software redundancy []. In the proposed scheme each submitted task T_i in the central queue is considered to be the primary task Pri_{T_i}. For each primary task Pri_{T_i} a replica(copy) of it is made which called the backup task $Back_{T_i}$. Both the primary and backup task are assigned to different randomly selected processors. The primary task is assigned to the primary queue of the selected processor, $Pri_{T_i} \rightarrow PQ_{P_k}$, Similarly the backup task is assigned to the backup queue of the other selected processor, $Back_{T_i} \rightarrow BQ_{P_l}$ where $k \neq l$. Hence Pri_{T_i} and $Back_{T_i}$ are assigned to different processor using Random allocation, i.e. $\forall Pri_{T_i} \in PQ_{P_k} \; \exists \; Back_{T_i} \in BQ_{P_l}$. The selection of the processor is checked for the overflow condition of the processor queue length. The tasks from the primary queue are executed using EDF uniprocessor scheduler, $Pri_{T_i} \in PQ_{P_k} \overset{EDF}{\rightarrow} P_k$. The backup task on other processor is updated periodically or when the last correct state of primary task is acquired. When a fault occurs and the primary task gets affected the backup task is moved to primary queue and is considered to be the primary task, $Back_{T_i} \in BQ_{P_l} \overset{Faulty\ Pri_{T_i} \in PQ_{P_k}}{\longrightarrow} Pri_{T_i} \in PQ_{P_l}$. The fault is detected by checking the intermediate deadline of the primary task. The fault tolerant technique has been given in Algorithm 2 illustrated in Figure 4.4.

Algorithm 2 (Fault Tolerant Algorithm).

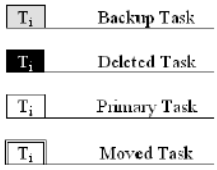
1: ***Input:****a setof periodictask set* $\xi = T_1, T_2, \ldots T_n$ *and a system resource set* Π
2: ***for*** $TIME = 1, 2, \ldots, sysc - 1, sysc$ ***do***
3: ***if*** $period(T_k) = TIME$ ***then***
4: *INSERT* T_k *to the Central Scheduler Queue*
5: ***end if***
6: ***if*** *Central Scheduler Queue* $\neq$ *NULL* ***then***
7: ***for*** *each Primary version* Pri_{T_i} ***do***
8: *Select the Non-Faulty end node* P_k
9: *INSERT the primary task to the* PQ_{P_k}
10: *provided its preceding tasks have executed*
(in case of precedence constrained task)
11: *Create a Backup version of task* $Back_{T_i}$
12: *Select the Non-Faulty end node* $P_l \neq P_k$
13: *INSERT the backup task to the* BQ_{P_l}

```
14:             end for
15:         end if
16:         if FAULT INJECTION ALGORITHM(System Resource Set) then
17:             FAULT INJECTION ALGORITHM(System Resource Set)
18:         end if
19:         for each processor P_i in the system do
20:             if new task added to the Primary Queue then
21:                 Rearrange the tasks in the Primary Queue of P_i according to EDF
22:                 if Deadline(T_Pi) > Deadline( Primary Queue[Front]) then
23:                     Preempt the currently executing task
24:                     Assign Primary Queue[Front] to the processor
25:                 end if
26:             end if
27:             Execute the task Pri_Ti assigned to Processor P_i from the Primary Queue PQ_Pi
28:             if Intermediate Deadline(T_i) exceeds TIME then
29:             Terminate the Pri_Ti
30:             Trigger a timing fault alarm
31:             Intimate the Back_Ti on remote processor P_j, i ≠ j
32:             end if
33:         end for
34:         Update Back_Ti to the last valid subtask of the Pri_Ti
35:         for each primary task which has failed do
36:             Rearrange tasks in the Backup Queue whose primary task has failed
                according to EDF
37:             DELETE Backup Queue[Front] and INSERT in Primary Queue
38:             Rearrange all tasks in the Primary Queue according to EDF
39:         end for
40: end for
```

The percentage increase in the throughput of FT Random-EDF than Faulty Random-EDF is 76% with upto 10% of permanent fault and arbitrary timing fault as shown in figure 4.8(a). The percentage increase in the throughput of FT Random-EDF than Faulty Random-EDF is 78% with upto 20% of permanent fault and arbitrary timing fault as shown in figure 4.8(b).

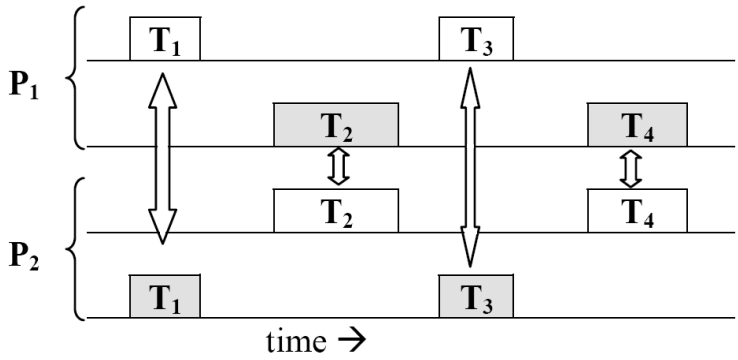
4.5 Adaptive Fault Tolerance Technique

The proposed adaptive technique uses the same principle as in Algorithm 2, except that primary task Pri_{T_i} is assigned to the best processor among to the two the has chosen for distribution of the primary and backup. $Back_{T_i}$ is assigned to the secondary processor.

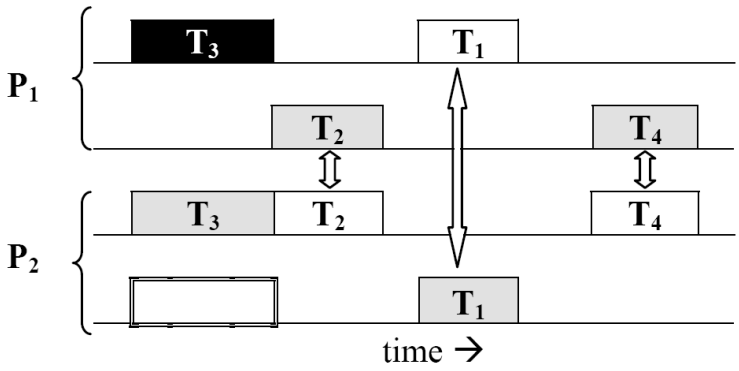


ID	Period	Computation Time	Deadline
1	3	12	8
2	5	32	13
3	7	41	2
4	11	8	15

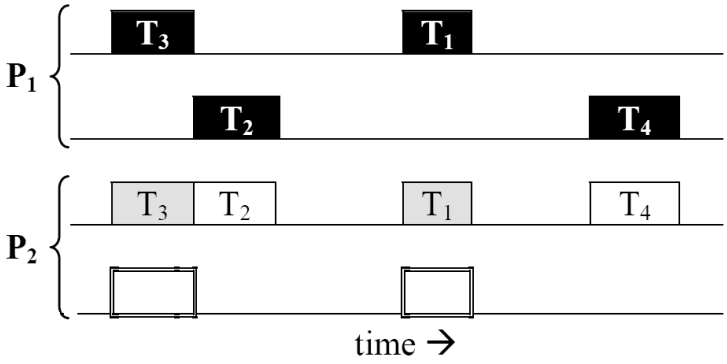
(a) Workload Parameter



(b) Initial Possible Schedule



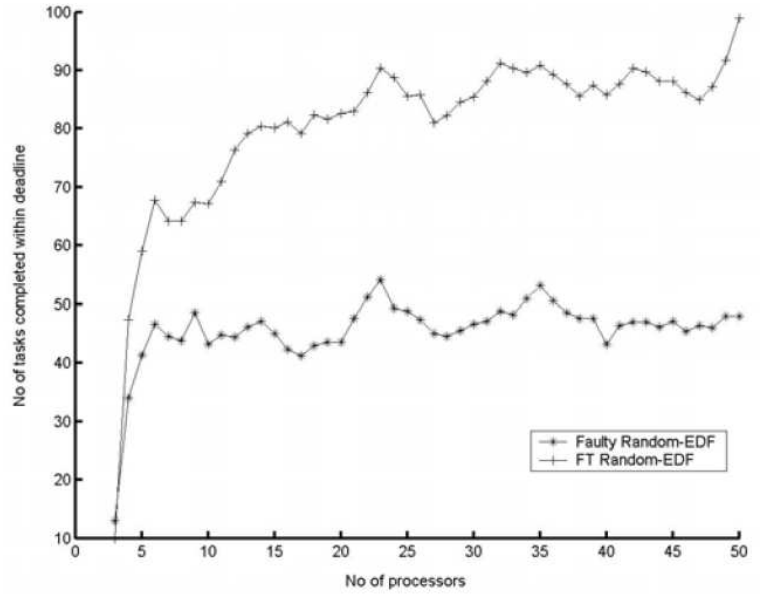
(c) When T_3 misses its deadline (Timing Fault) At $Time = t_n$



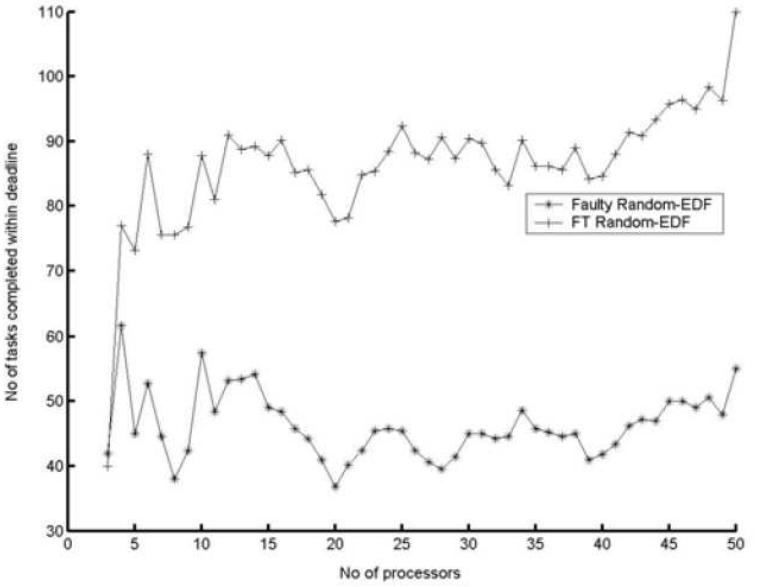
(d) When P_1 encounters a Permanent fault At $Time = t_n$

Figure 4.4: Schematic view of proposed fault tolerant technique

If the task misses its deadline or the processor on which the primary task has been assigned crashes, the backup tasks are shifted to the primary queue of the same node. When the backup task has been added to primary queue an additional replica of the



(a) Performance of Fault tolerant technique in presence of permanent fault $N_{FP} < 10\%$ and timing fault.



(b) Performance of Fault tolerant technique in presence of permanent fault $N_{FP} < 20\%$ and timing fault.

Figure 4.5: Simulation Results for independent tasks

backup task is generated and distributed to another node in the system. The Adaptive Fault Tolerant Algorithm has been mentioned in 3. The System's communication links are assumed to be fault-free. The adaptive technique is illustrated in figure 4.6.

The performance of the proposed methods in presence of 10% and 20% permanent fault and arbitrary timing fault for precedence constrained tasks are shown in Figure 4.7(a) and Figure 4.7(b) respectively. The results Figure 4.8(a)- 4.8(b) show that our adaptive FT algorithm FT Random-EDF heuristic and Faulty Random-EDF heuristic under 10% and 20% of permanent fault respectively and arbitrary number of timing

and transient fault of precedence constrained tasks.

Algorithm 3 (Adaptive Fault Tolerant Scheduling Algorithm).

1: ***Input:****a setof periodictask set* $\xi = T_1, T_2, \ldots T_n$ *and a system resource set* Π
2: ***for*** $TIME = 1, 2, \ldots, sysc - 1, sysc$ ***do***
3: ***if*** $period(T_k) = TIME$ ***then***
4: *INSERT* T_k *to the Central Scheduler Queue*
5: ***end if***
Choose the primary and backup among two versions of the task
6: ***if*** *Central Scheduler Queue* $\neq$ *NULL* ***then***
7: *Select the Non-Faulty node* P_k
8: *Select the Non-Faulty node* $P_l \neq P_k$
Compare the Total Waiting Time (TWT) for P_k ***and*** P_l
9: *PTR=*P_i*[Front]*
10: ***while*** *Primary Queue*(P_i)*[PTR]≠Primary Queue*(P_i)*[REAR]* ***do***
11: $TWT(P_i) = TWT(P_i) + PQ_{P_i}[PTR]/Execution\ Rate(P_i)$
12: *INCREMENT PTR*
13: ***end while***
14: ***if*** $TWT(P_k) < TWT(P_l)$ ***then***
15: *INSERT task* Pri_{T_i} *to the Primary Queue of* PQ_{P_k}
16: *INSERT* $Back_{T_i}$ *to the Backup Queue of* BQ_{P_l}
17: ***else***
18: *INSERT task* Pri_{T_i} *to the Primary Queue of* PQ_{P_l}
19: *INSERT the* $Back_{T_i}$ *to the Backup Queue of* PQ_{P_k}
20: ***end if***
21: ***end if***
22: ***if*** *FAULT INJECTION ALGORITHM(System Resource Set)* ***then***
23: *FAULT INJECTION ALGORITHM(System Resource Set)*
24: ***end if***
25: ***for*** *each processor* P_i *in the system* ***do***
26: ***if*** *new task added to the* PQ_{P_i} ***then***
27: *Rearrange the tasks in the* PQ_{P_i} *according to EDF*
28: ***if*** $Deadline(T_{P_i}) > Deadline($ *Primary Queue[Front]*$)$ ***then***
29: *Preempt the currently executing task*
30: *Assign Primary Queue[Front] to the processor*
31: ***end if***
32: ***end if***
33: *Execute the task* T_i *assigned to Processor* P_i *from the Primary Queue* PQ_{P_i}
34: ***if*** *Intermediate Deadline*(T_i) *exceeds TIME* ***then***
35: *Terminate the Primary version of task* T_i
36: *Trigger a timing fault alarm*

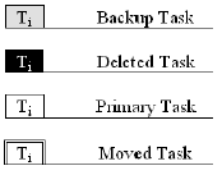
37: *Intimate the Backup version of task $Back_{T_i}$ on remote processor P_j, $i \neq j$*
38: ***end if***
39: ***end for***
40: *Update $Back_{T_i}$ to the last valid subtask of the Pri_{T_i}*
41: ***for*** *each primary task which has failed* ***do***
42: *Rearrange tasks in the Backup Queue whose primary task has failed according to EDF*
43: *DELETE Backup Queue[Front] and INSERT in Primary Queue*
44: *Rearrange all tasks in the Primary Queue according to EDF*
45: *select $P_j \neq P_i$*
46: *INSERT the backup version of Backup Queue[Front] to the BQ_{P_j}*
47: ***end for***
48: ***end for***

The increase in the guarantee ratio of FT Random-EDF than Faulty Random-EDF is 30% whereas the guarantee ratio by the adaptive technique gave 99% guarantee to finish tasks even in presence of permanent fault of upto 10% of faulty system and arbitrary number of timing faults as shown in figure 4.7(a) and figure 4.7(b) for 20% of permanent fault and arbitrary number of timing fault for precedence constrained periodic tasks.

We have investigated our approach against arbitrary number of transient in addition to permanent and timing fault. The percentage increase in the guarantee ratio of FT Random-EDF than Faulty Random-EDF is 15% while increase in the guarantee ratio of ADFT Random-EDF than FT Random-EDF is 67% in presence of upto 10% of guarantee ratio fault shown in figure 4.8(a). In presence of permanent fault upto 20% the percentage increase in the guarantee ratio of FT Random-EDF than Faulty Random-EDF is 12.78% while increase in the guarantee ratio of ADFT Random-EDF than FT Random-EDF is 53.18% shown in figure 4.8(b).

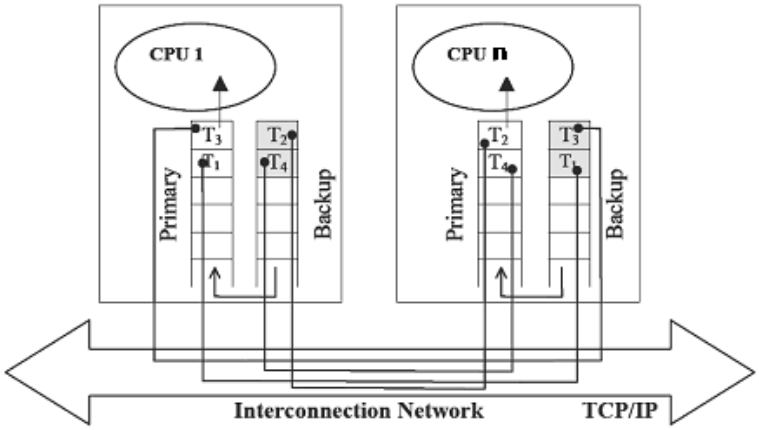
4.6 Conclusion

The performance measure presented here will help the distributed computing community in the implementation of resources management with random periodic traffic. This technique needs further investigation on other type of distributed system faults that affects the system performance.

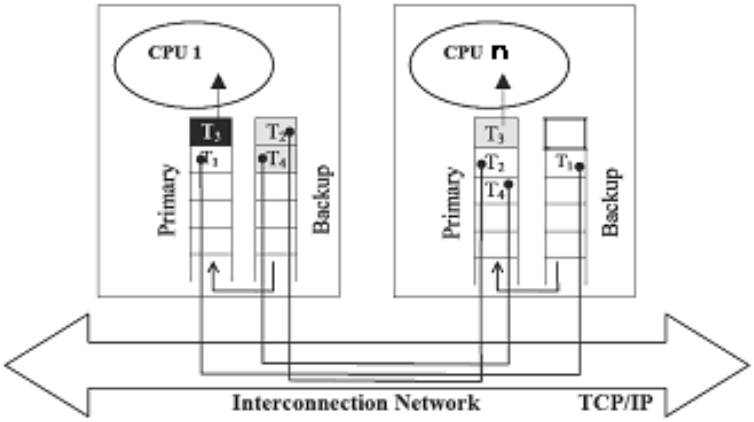


ID	Period	Computation Time	Deadline
1	3	12	8
2	5	32	13
3	7	41	2
4	11	8	15

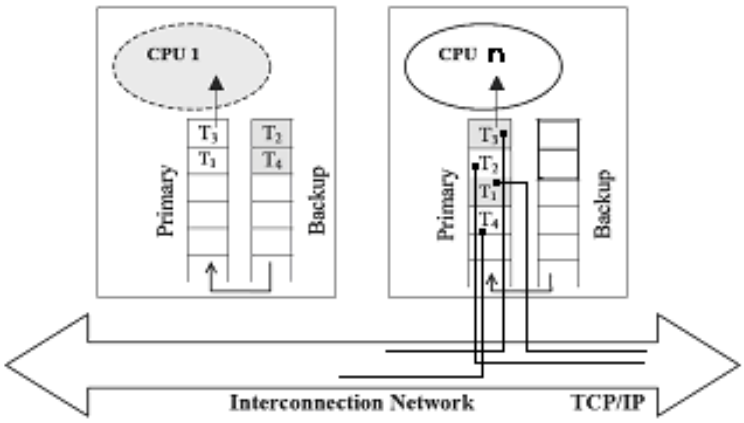
(a) Workload Parameter



(b) Initial Possible Schedule

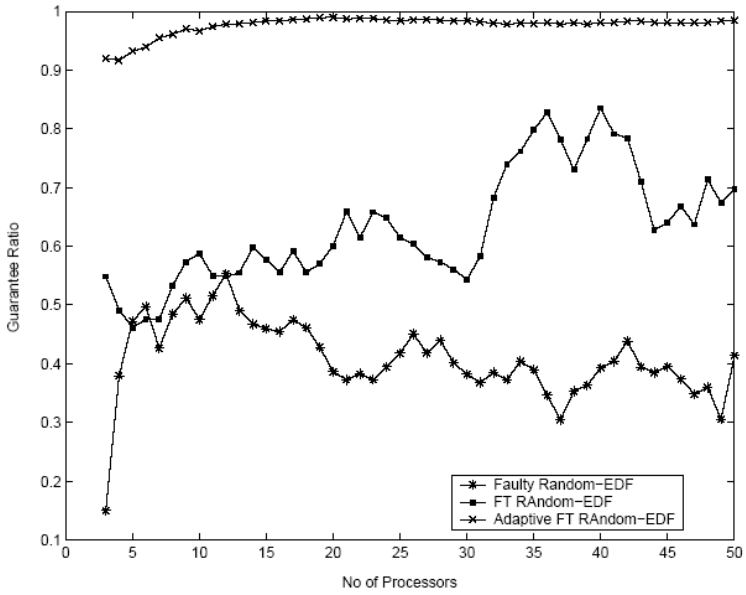


(c) When T_3 misses its deadline (Timing Fault) At $Time = t_n$

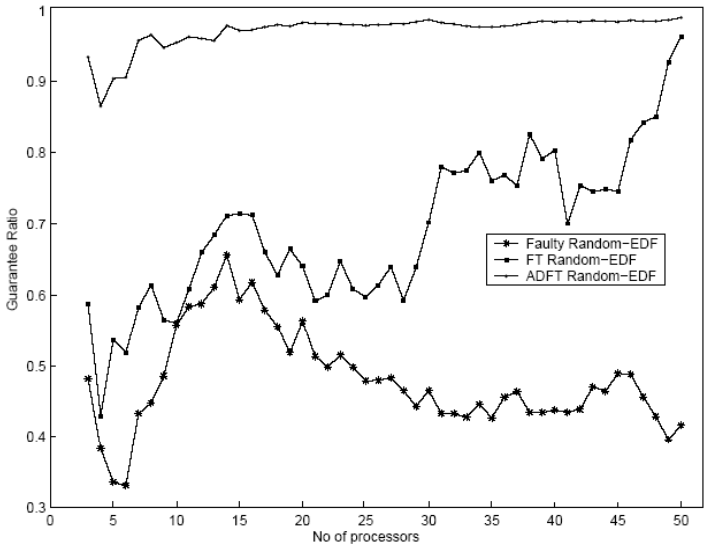


(d) When P_1 encounters a Permanent fault At $Time = t_n$

Figure 4.6: Schematic view of proposed adaptive fault tolerant technique

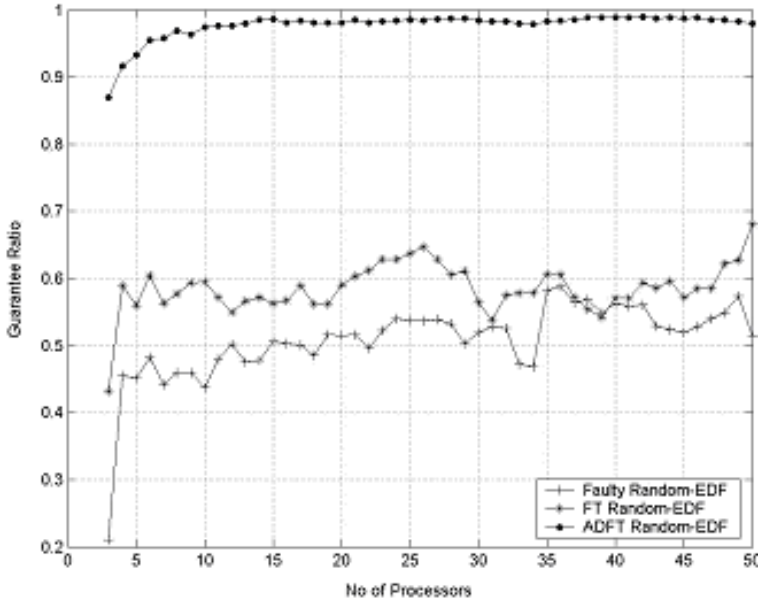


(a) Performance of Fault tolerant technique in presence of N_{FP} < 10% and timing fault.

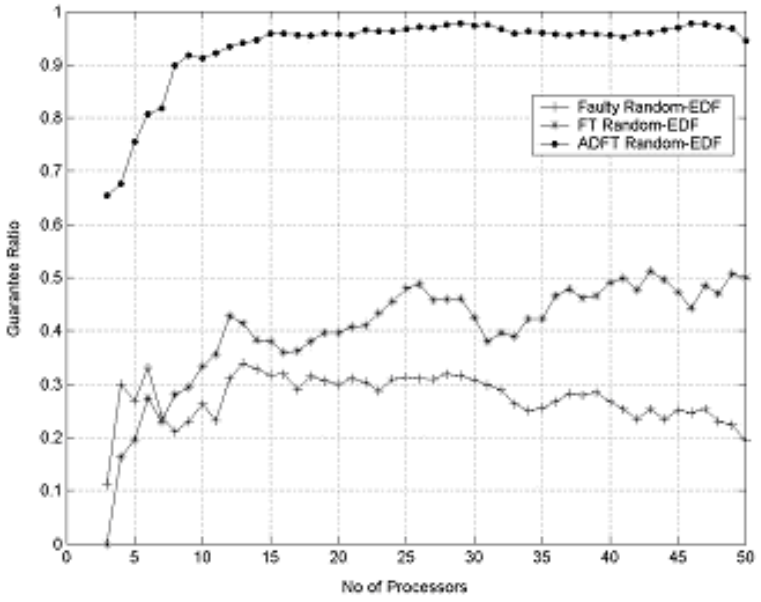


(b) Performance of Fault tolerant technique in presence N_{FP} < 20% and timing fault.

Figure 4.7: Simulation Results for tasks with precedence constraints



(a) Comparison of Fault tolerant technique and Adaptive Fault tolerant technique against the Faulty Random-EDF in presence of permanent and transient fault i.e. $N_{FP} \leq 10\%$ and arbitrary timing and transient fault.



(b) Comparison of Fault tolerant technique and Adaptive Fault tolerant technique against the Faulty Random-EDF in presence of permanent and transient fault i.e. $N_{FP} \leq 20\%$ and arbitrary timing and transient fault.

Figure 4.8: Simulation Results for comparison for Fault Tolerant and Adaptive Fault Tolerant Technique

Chapter 5

Task scheduling using Evolutionary Computing

5.1 Introduction

Major area of evolutionary computing research focuses on a set of techniques inspired by the biological sciences, because biological organisms often exhibit properties that would be desirable in computer systems. In the state-of-art evolutionary optimization techniques, computing is processing with mimic organism evolution process. Evolutionary computation uses iterative process based upon various techniques inspired by biological mechanism of evolution. These techniques are genetic algorithms (GA), genetic programming (GP), and classifier systems (CS), DNA computing, particle swarms, ant colonies etc. Early attention has focused on DNA because its properties are extremely attractive as a basis for a computational system. The code of DNA is essentially a digital code, particular strands of DNA can be used to code information, and in particular, joining and other recombinations of these strands can be used to represent putative solutions to certain computational problems. On the other hand, in DNA computing, because the computing mimics DNA copy mechanism in chemical reaction, it processes massive parallel.

DNA computing is being established as a viable alternative to solve various NP-complete problems. These alternative approaches to performing exhaustive search in solution space are found to be more efficient for various intractable problems. The computer and the DNA both use information embedded in simple coding, the binary software code and the quadruple genomic code, respectively, to support system operations. On top of the code, both systems display a modular, multi-layered architecture, which, in the case of a computer, arises from human engineering efforts through a combination of hardware implementation and software abstraction. A process represents just sequentially ordered actions by the CPU and only virtual parallelism can be implemented through CPU time-sharing. Whereas process management in a com-

puter may simply mean job scheduling, coordinating pathway bandwidth through the gene expression machinery represents a major process management scheme in a DNA. In summary, a DNA can be viewed as a super-parallel computer, which computes through controlled hardware composition. A comparison between the architecture of a computer and a cell, within which DNA resides, has been given in Figure 5.1.

Nowadays the research effort in the area of DNA computing concentrates on four main problems: designing algorithms for some known combinatorial problems, designing new basic operations of "DNA computers", developing new ways of encoding information in DNA molecules and reduction of error in DNA based computations [27]. So, we are inspired to use DNA computing to solve the assignment and scheduling problem in distributed system, which is a NP-Hard problem. In this chapter DNA based algorithms for solving the task assignment problem on real time distributed system is presented. To our best knowledge it is the first attempt to solve this problem by molecular algorithms.

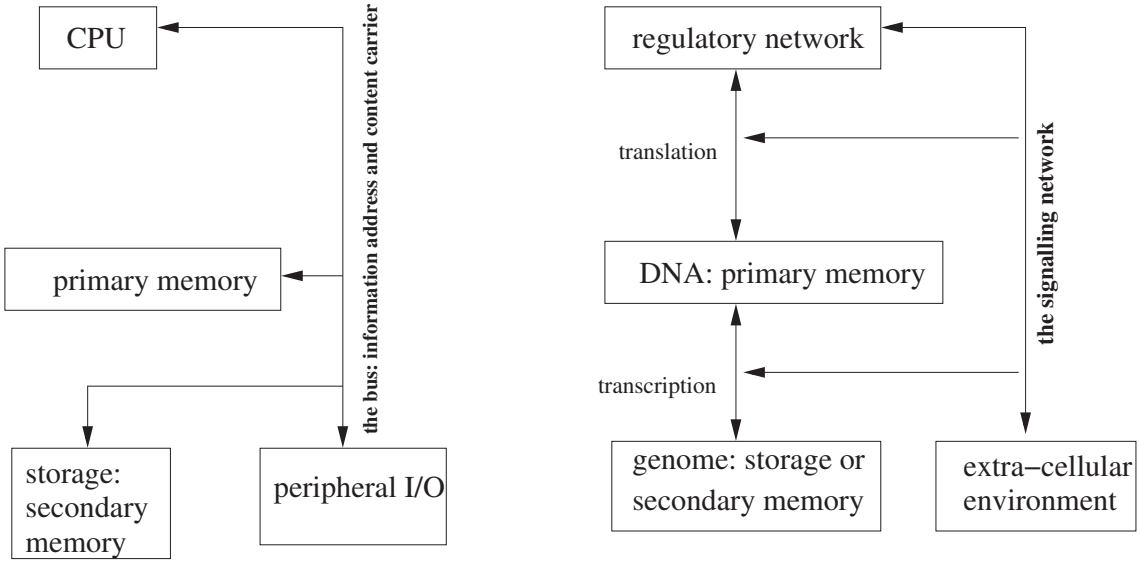


Figure 5.1: A simplistic schematic comparison of the architecture of a computer and a cell. Note that RNA acts as a messenger between DNA

The general scheme of an evolutionary algorithm can be given as in Algorithm 4 in the pseudo-code fashion [25].

Algorithm 4. *The general scheme of an evolutionary algorithm in pseudo-code*

BEGIN

INITIALISE population with random candidate solutions;
EVALUATE each candidate;
REPEAT UNTIL (TERMINATION CONDITION is satisfied) DO
1. SELECT parents;
2. RECOMBINE pairs of parents;

3. MUTATE the resulting offspring;
4. EVALUATE new candidates;
5. SELECT individuals for the next generation;
OD
END

We propose a DNA coding model to solve real time scheduling problem in real time distributed system.

5.2 DNA Computing

DNA computing is to use DNA as the platform to compute by means of molecular biology techniques for encoding information, generating potential solutions, and selecting and identifying correct solutions [92]. DNA computing, also known as molecular computing, is a new approach to massively parallel computation based on ground breaking work by Adleman.

DNA (deoxyribonucleic acid) is a double stranded sequence of four nucleotides; the four nucleotides that compose a strand of DNA are as follows: adenine (A), guanine (G), cytosine (C), and thymine (T); they are often called bases. The chemical structure of DNA (the famous double- helix) was discovered by James Watson and Francis Crick in 1953. It consists of a particular bond of two linear sequences of bases. This bond follows a property of complementarity: adenine bonds with thymine (A-T) and vice versa (T-A), cytosine bonds with guanine (CG) and vice versa (G-C). This is known as Watson-Crick complementarity. Each DNA strand has two different ends that determine its polarity: the 3.'end, and the 5.'end. The double helix is an anti-parallel (two strands of opposite polarity) bonding of two complementary strands.

DNA computers work by encoding the problem to be solved in the language of DNA: the base-four values A, T, C and G. Using this base four number system, the solution to any conceivable problem can be encoded along a DNA strand like in a Turing machine tape. Every possible sequence can be chemically created in a test tube on trillions of different DNA strands, and the correct sequences can be filtered out using genetic engineering tools.

There are four reasons for using molecular biology to solve computational problems.

1. **The information density of DNA is much greater than that of silicon :** 1 bit can be stored in approximately one cubic nanometer. Others storage media, such as videotapes, can store 1 bit in 1,000,000,000,000 cubic nanometer.

2. **Operations on DNA are massively parallel :** a test tube of DNA can contain trillions of strands. Each operation on a test tube of DNA is carried out on all strands in the tube in parallel

3. **DNA computing is an interdisciplinary field where :** biologists, computer scientists, physics, mathematicians, chemists, etc. find a lot of interesting problems which can be applied to both theoretical and practical areas of DNA computing.

4. **Error tolerance capability :** the DNA exhibit much better error tolerance capability, resulting in improved system robustness. Redundancy plays a major role. The DNA have multiple mechanisms to process the same information. If one fails, there are other ways to ensure that crucial cellular processes, such as programmed cell death, are completed [111].

5.3 DNA Computational Model

DNA replication techniques are used to solve NP complete problems. Adleman in 1994 solved the TSP problem using DNA solution technique [3]. The technique uses the possible combinations of valid routes among the cities by Marge and anneals the two-solution set. One of the solutions is the cities DNA molecule and the other as edge DNA molecule.

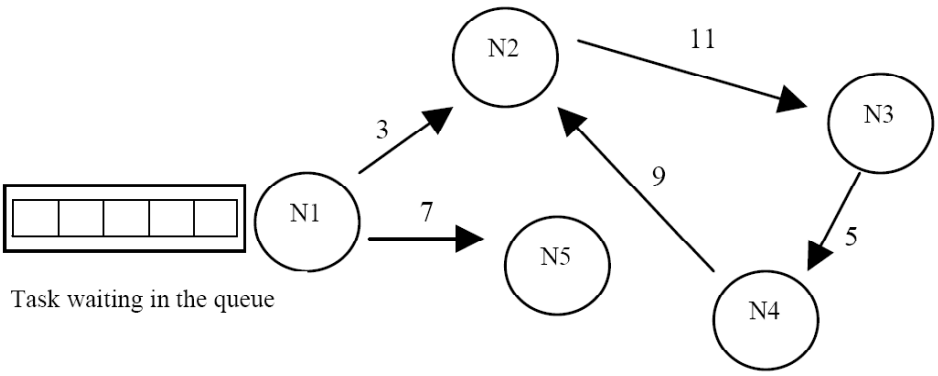


Figure 5.2: Schematic structure of RTDS

In this section a DNA replication technique using fixed length coding to select the computing node to which task is to be assigned in RTDS as shown in Figure 5.2 to execute a task generated on one site with real time constraints, has been proposed. The approach to this work starts by relating each task on the source computing node a salesman [3] and that after the destination node is selected the task is transferred to that site. Table 5.1 shows the city and cost fixed length code for five nodes and costs in RTDS [99].

5.3.1 Encoding

The encoding scheme, which includes the communication cost between a given pair of computing nodes i.e., source and destination is presented in [99, 58]. Table 5.1 shows

the encoded values for the given RTDS as shown in Figure 5.2.

Node	Sequence
1	A—G—T—C—G—G
2	G—T—G—G—A—C
3	C—A—G—T—A—A
4	T—A—T—G—G—G
5	A—A—G—G—C—C
Cost	
3	A—T—G—A—T—A
5	G—G—A—T—A—A
7	T—A—C—T—A—A
9	C—C—T—G—C—A
11	T—T—A—C—C—A

Table 5.1: Node and cost sequences for the five node RTDS

5.3.2 Operations

The basic operations of DNA algorithm are usually designed for selecting which satisfy some particular conditions. On the other hand there may be different sets of such basic operations. The set of operations used in this problem are:

1. MERGE: given test tube TT_1 and TT_2, create a new tube TT containing all strands form TT_1 and TT_2.

2. AMPLIFY: given tube TT create copy of them.

3. DETECT: given tube TT return *true* if TT contains at least one DNA strand, otherwise return $false$.

4. SEPARATE: given tube TT and word w over alphabet $\sum_{DNA}$ create two tubes $+(TT, w)$ and $-(TT, w)$, where $+(TT, w)$ consists of all strand from TT containing w as a substring and $-(TT, w)$ consists of the remaining strands.

5. LENGTH-SEPARATE: given tube TT and positive integer n create tube $(TT, \leq n)$ containing all strands from TT which are of length n or less.

6. POSITION-SEPARATE: given tub TT and word w over alphabet $\sum_{DNA}$ create tube $B(TT, w)$ containing all strands from TT which have w as prefix and tube $E(TT, w)$ containing all strands from TT which have w as suffix.

Each of the above operations is a result of some standard biochemical procedure.

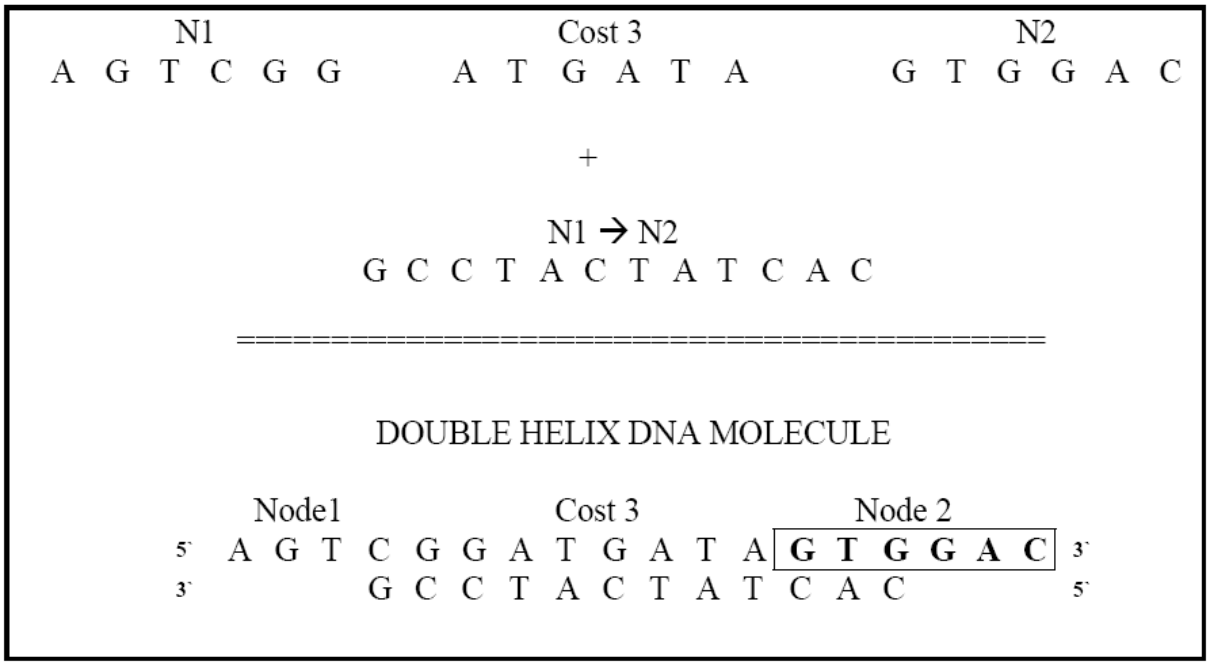


Figure 5.3: DNA Computation for N1→ N2 including the cost.

5.4 Proposed Algorithm for Task Assignment

Tasks in real time systems can be either periodic or aperiodic in nature. In real time systems the functions not just to be logically correct but also to be within deadline. Tasks are said to be periodic when they arrive within a fixed interval of time. The periodic real time tasks can be described by its period, deadline and the worst case execution time. Details about the workload model has been given in Chapter 2.3. In this section we propose a task assignment/allocation scheme for RTDS.

Task Assignment using DNA is done in the following four basic steps:

Algorithm 5. *Task Assignment using DNA*

1: ***Solution Space Generation Phase.*** *Encode each computing node and the cost of communication with 6 base strands and the edge between each node with complementary base. Create copies of them.*
2: ***Select itineraries that start and end with the correct nodes.*** *Select strands, which have start node as the centralized scheduler and end nodes as the possible connected destination node.*
3: ***Select itineraries that contain the least cost of communication.*** *Check the cost of the DNA strand by decoding the code sequence of the cost between the source and destination node.*
4: ***Select itineraries that give feasible schedule.*** *Remove the last six codes of the top strand i.e. node N2. Check the feasibility of sending the task if assigned to it depending on available resources such as available CPU time, deadline constraint of the task which may or may not depend on the period of the task. If the choice is feasible then the*

assignment is made else go for the next available strand.

This proposed algorithm is used to design a fault tolerant real-time scheduling scheme for hard real time task. This scheme results multiple feasible allocation schemes, with out computational overhead in comparisons to other iterative evolutionary techniques. The proposed scheme is outlined in Figure 5.4 (darkened block) as follows:

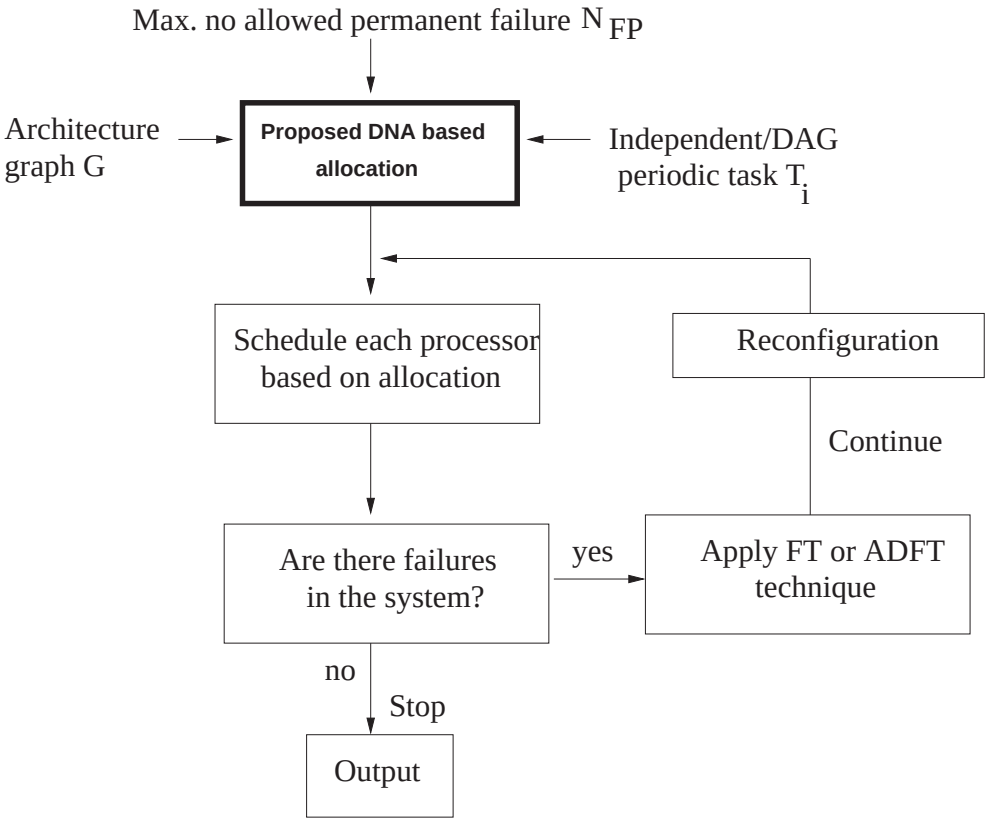


Figure 5.4: Developing a DNA Task allocation based Fault Tolerant RTDS schedule

5.5 Conclusion

The research in DNA computing is in a primary level. High information density of DNA molecules and massive parallelism involved in the DNA reactions make DNA computing a powerful tool. Tackling problems with DNA computing would be more appropriate when the problems are computationally intractable in nature. The basic effort is how efficiently the problem is being represented using DNA. This is always a open problem for the researchers, to represent a problem so that, each iteration yield results solutions for the said problem. Because the DNA Computing due to its high degree of parallelism, can overcome the difficulties that may cause the problem intractable on silicon computers. However using DNA computing principles for solving simple problems may not be suggestible. It has been proved by many

research accomplishments that any procedure that can be programmed in a silicon computer can be realized as a DNA computing procedure.

In this chapter, we have proposed a new framework for assigning task in heterogeneous RTDS including the cost of path connecting the nodes.

Chapter 6

Thesis Conclusion

Fault-tolerance becomes an important key to establish dependability in real time distributed system (RTDS). Hardware and software redundancy are well-known effective methods for hardware fault-tolerance, where extra hard ware (e.g., processors, communication links) and software (e.g., tasks, messages) are added into the system to deal with faults.

In a real time distributed system scheduling strategies determine which computational tasks will be executed on which processors at what times, allocating the task to one processor in the system using a specific strategy basically carries this out. Number of such strategy has been reported in the literature for dynamic task allocation heterogeneous systems, or developing frameworks for evaluating such strategies. Scheduling involves the allocation of resources and time to tasks in such a way that certain performance requirements are met. It is believed that the basic problem in real-time systems is to make sure that tasks meet their time constraints. Scheduling is also a well-structured and conceptually demanding problem. In addition to timing and predictability constraints, tasks in a real-time application also have other constraints one normally sees in traditional non-real-time applications

The scheduling problem in real time distributed systems can be conceptually separated into two parts, as task allocation, or global scheduling and local scheduling or uniprocessor scheduling. Ordinary scheduling algorithms attempt to ensure fairness among tasks, minimum progress for any individual task, and prevention of starvation and deadlock. To the scheduler of real-time tasks, these goals are often superficial. The primary concern in scheduling real-time tasks is deadline compliance. The performance of two batch mode (max-min, min-min) and two immediate mode (earliest first, lightest load) scheduler with variable loads has been simulated using our simulator. We have studied the effect of faults in real time distributed system using these above batch mode and immediate mode schedulers. The resulted observation has been used to design the proposed fault tolerant scheduling schemes.

Research in fault-tolerant distributed computing, aims at making distributed sys-

tems more reliable by handling faults in complex computing environments. Moreover, the increasing dependence of different services on real time heterogeneous distributed system has led to an increasing demand for dependable systems, systems with quantifiable reliability properties. Task scheduling techniques can be used to achieve effective fault tolerance in real time systems. This is an effective technique, as it requires very little redundant hardware resources. Fault tolerance can be achieved by scheduling additional ghost copies in addition to the primary copy of the task. We have proposed new algorithms based on software redundancy to tolerate permanent and timing failures. We have considered heterogeneous real time distributed systems with point-to-point communication links for this research work. These proposed algorithms are resigned to meet two basic objectives (i) the execution of backup versions should not hinder the execution of the primary version of the tasks, and (ii) when the primary task fails to meet its deadline the backup instance should then be executed but it should be executed from the point of last correct sub task executed by the primary version.

The proposed algorithm uses distributed recovery block to perform software redundancy, where a given input a task (T_i) is augmented with a redundancies. Then, operations and data-dependences of T_i can be distributed and scheduled on a specified target distributed architecture (G) to generate a fault tolerant distributed schedule. A performance analysis has been presented considering the guarantee ratio of different algorithm, mentioned in Fault Tolerant Algorithm 2 and Adaptive Fault Tolerant Algorithm mentioned in Algorithm 3. The performance of all the proposed schemes has been verified using our own simulator.

Tackling problems with DNA computing would be more appropriate when the problems are computationally intractable in nature. The basic effort is how efficiently the problem is being represented using DNA. We have proposed a new framework for assigning task in heterogeneous RTDS including the cost of path connecting the nodes. The proposed approach uses DNA replication technique using fixed length coding to select the computing node to which task is to be assigned in RTDS.

The development in the thesis is genuinely supported detail literature survey and mathematics preliminaries leading to the proposed model. For shake of continuity each chapter has its relevant introduction and theory. The work is also supported by list of necessary references. Attempt is made to make the thesis self-content.

Chapter 7

Future Works

Fault-tolerance is an important requirement for real-time distributed system, which is designed to provide solutions in a stringent timing constraint. We have considered both fault-tolerant and adaptive fault-tolerant scheme on heterogeneous multi-component distributed system architecture using a software technique based on Distributed Recovery Block (DRB). Apart from these faults, Real time distributed systems may be effected by the other faults such as Omission and Byzantine fault. Impact of these less significant faults in real-time system need to be investigated using the proposed scheme.

Also, Fault Tolerant problems for which hardware solutions of moderate costs exist but pure software solutions are sought merely because if found, they can be more cost effective under the current hardware economy, should be viewed as short-term research problems. A key objective of our future work is to conduct further experiments on implementations of commercial fault-tolerant and real-time protocols using the CORBA environment.

Distributed systems are becoming a popular way of implementing many embedded computing applications, automotive control being a common and important example. Such embedded systems typically have soft or hard performance constraints. The increasing complexity of these systems makes them vulnerable to failures and their use in many safety-critical applications makes fault tolerance an important requirement. Embedded systems account for a major part of critical applications (space, aeronautics, nuclear) as well as public domain applications (automotive, consumer electronics). Their main features are: duality automatic control/discrete-event, critical real-time, limited resources, and distributed and heterogeneous architectures.

Reliability and availability are only two of the many attributes that fault-tolerant embedded systems must have. Users also demand dependability, safety, and security, and these elements are an integral part of embedded-system operation. Hardware and software must meet ever changing and stringent requirements to remain readily adaptable to a multitude of applications and environments. Researchers in this field

generally acknowledge that embedding fault tolerance in embedded systems requires a radical change in the overall design process. Practical solutions require a full understanding of the hardware and software domains, their relationships in an embedded environment, and system behavior under fault-free and faulty conditions. It would be interesting to study how much dependability the proposed schemes and ideas, developed in this thesis work, could provide to such systems.

Periodic and aperiodic tasks co-exist in many real-time systems. The periodic tasks typically arise from sensor data or control loops, while the aperiodic tasks generally arise from arbitrary events. Their time constraints need to be met even in the presence of faults. Scheduling both periodic and aperiodic tasks in real-time systems is more difficult than scheduling periodic or aperiodic tasks alone. The simplest method is to treat the aperiodic tasks as background tasks when their response times are not critical. However the performance of DRB based fault tolerant scheduling scheme can be further studied considering both periodic and aperiodic tasks in real-time systems

We have explored the possibility of ensuring fault tolerance in the real time system without any communication cost involved in task scheduling. Real-time communication is concerned with the problem of delivering messages by their deadlines. Such a communication service with delay guarantees is crucial for mission critical real-time applications. If systems are geographically separated and computing nodes are sharing a non-dedicated communication network, then the communication cost contributes to the completion time of a task executed on this environment. Our proposed method needs further investigation considering extra time [selection of target computing node + time to communicate the data + time to communicate the task] in addition to the expected execution time. Fault tolerant real-time communication deals with delivery of messages by their deadlines even in the presence of faults. It also suggests exploring the applicability of these techniques to mission critical applications in embedded systems (such as those in submarine, aircraft, or industrial process controllers) demand Quality of Service (QoS) guarantees in terms of bounded message transfer delays.

Autonomic computing basically based upon self-healing property, which operates, on additional hardware requirement. Autonomous Systems and their embedded autonomy software in many application perform correctly for an extended period of time and make real-time decisions that meet both logical and timing requirements. Recent development in autonomic computing research, open a new frontier to study the system behavior in autonomic computing environment. Feasibility of applying the proposed methods may be considered for Autonomous Real time system.

In Real Time Fault Tolerant Distributed Computing systems, fault detection and recovery actions should ideally be executed such that intended output actions of Real Time computations always take place on time in spite of fault occurrences. When it is not feasible, the fault tolerance actions which lead to the least damages to the application missions / users must be attempted. The liveliness of the Fault Tolerant Distributed Computation field is in an upward move at this juncture of 21^{st} century. A

basic technical foundation for Fault Tolerant Distributed Computing was laid out in the last quarter of 20^{th} century. It contains among others basic techniques for fault detection and network surveillance, transaction structuring and execution, checkpointing and rollback, and replicated processing and recovery.

Bibliography

[1] C. Ward A. Y. Zomaya and B. Macey. Genetic scheduling for parallel processor systems: comparative studies and performance issues. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):795812, August 1999. [cited at p. 60]

[2] Tarek F. Abdelzaher and Kang G. Shin. Combined task and message scheduling in distributed real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(11):1179–1191, November 1999. [cited at p. 59]

[3] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, November 1994. [cited at p. 82]

[4] Marcos K. Aguilera1, Gerard Le Lann, and Sam Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. *Springer-Verlag DISC 2002*, pages 354 – 369, 2002. [cited at p. 5, 27]

[5] L. Alvisi, E. Elnozahy, S. Rao, S.A. Husain, and A. de Mel. An analysis of communication induced checkpointing. In *29th Annual International Symposium on Fault-Tolerant Computing*, pages 242 – 249. IEEE, June 1999. [cited at p. 29]

[6] Darin Anderson. Providing fault tolerance in distributed systems. *The Computer Science Discipline Seminar Conference*, 2000. [cited at p. 5]

[7] Anish Arora and Mohamed G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *Software Engineering*, 19(11):1015–1027, 1993. [cited at p. 3, 19]

[8] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: A taxonomy. *IFIP World Computer Congress*, August 2004. [cited at p. 4]

[9] Babaoglu, Ozalp, and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993. [cited at p. 27]

[10] J.A. Bannister and K.S. Trivedi. Task allocation in fault-tolerant ditributed systems. *Acta Informatica*, 20:261–281, 1983. [cited at p. 57]

[11] V. Berten, J. Goossens, and E. Jeannot. A probabilistic approach for fault tolerant multiprocessor real-time scheduling. *20th International Parallel and Distributed Processing Symposium*, pages 8–16, April 2006. [cited at p. 4, 43]

[12] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and J. Xu. An adaptive approach to achieving hardware and software fault tolerance in a distributed computing environment. *Journal of Systems Architecture*, 47:763781, 2002. [cited at p. 43]

[13] Tom Bracewell and Priya Narasimhan. A middleware for dependable distributed real-time systems. 2001. [cited at p. 43]

[14] A. Burchard, J. Liebeherr, Y. Oh, and S.H. Son. New strategies for assigning real time tasks to multiprocessor systems. *IEEE Transaction On Computers*, 44(12):1429–1442, December 1996. [cited at p. 57]

[15] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering*, 6(3):116–128, May 1991. [cited at p. 5]

[16] George Candea. *The Basics of Dependability*. Stanford University. [cited at p. 24]

[17] Changgui Chen, Weijia Jia, and Wanlei Zhou. A reactive system architecture for building fault-tolerant distributed applications. *The Journal of Systems and Software*, 72:401415, 2004. [cited at p. 43]

[18] S. Chodrow, F. Jahanian, and M. Donner. Runtime monitoring of real-time systems. *IEEE Real-Time Systems Symposium*, pages 74–83, December 1991. [cited at p. 5]

[19] E.G. Coffman. *Computer and Job-Shop Scheduling Theory*. Wiley, New York, 1976. [cited at p. 57]

[20] E.G. Coffman and M.R. Garey. Proof of the 4/3 conjecture for preemptive vs nonpreemptive two processor scheduling. *Journal of ACM*, 40(5), 1993. [cited at p. 58]

[21] Eric C. Cooper. Replicated distributed programs. In *Proceedings of the tenth ACM symposium on Operating systems principles, ACM SIGOPS Operating Systems Review*. ACM Press, December 1985. [cited at p. 41]

[22] Flavin Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991. [cited at p. 5, 21, 63]

[23] S. Davari and S.K. Dhall. An online algorithm for real time tasks allocation. *Proceedings of IEEE Real Time Systems Symposium*, pages 194–200, 1986. [cited at p. 57]

[24] S.K. Dhall and C.L. Liu. On a real time scheduling problem. *Operation Research*, 26(1):127–140, February 1978. [cited at p. 57]

[25] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*, chapter What is an Evolutionary Algorithm? Natural Computing Series. Springer ISBN 3-540-40184-9, 2003. [cited at p. 80]

[26] Gerhard Fohler and Krithi Ramamritham. Static scheduling of pipelined periodic tasks in distributed real time systems. Department of Computer Science, University of Massachusetts. [cited at p. 59]

[27] Piotr Formanowicz. Dna computing. *Computational Methods in Science and Technology*, 11(1):11–20, 2005. [cited at p. 80]

[28] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co, New York, 1979. [cited at p. 51]

[29] M.R. Garey and D.S. Johnson. *Computing and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979. [cited at p. 47]

[30] S. Ghosh, R. Melhem, and D. Mosse. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):272–284, March 1997. [cited at p. 4, 43]

[31] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. Generation of fault-tolerant static scheduling for real-time distributed embedded systems with multi-point links. *Proceedings 15th International Parallel and Distributed Processing Symposium*, pages 1265 – 1272, April 2001. [cited at p. 4, 43]

[32] Joel Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. Technical Report TR01-024, 14 2001. [cited at p. 5]

[33] Felix C. Grtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1), March 1999. [cited at p. 4, 19]

[34] I. Gupta, G. Manimaran, and C. Siva Ram Murthy. Primary-backup based fault-tolerant dynamic scheduling of tasks in multiprocessor real-time systems. *Proceedings of IEEE Fault-Tolerant Computing Symposium*, June 1999. [cited at p. 4, 40, 43, 65]

[35] D. Haban and K.G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. *IEEE Transactions on Software Engineering*, 16(12):1374 – 1389, December 1990. [cited at p. 5]

[36] W. L. Heimerdinger and C.B. Weinstock. A conceptual framework for system fault tolerance. Technical Report CMU/SEI-92-TR-33, Software Engineering Institute, Carnegie Mellon University, Pennsylvania, October 1992. [cited at p. 21, 63]

[37] Walter L. Heimerdinger and Chuck B. Weinstock. A conceptual framework for system fault tolerance. Technical Report CMU/SEI-92-TR-33, Carnegie Mellon University, Pittsburgh, PA, 1992. [cited at p. 4]

[38] Y.S. Hong and H.W. Goo. A fault tolerant scheduling for periodic tasks in real time systems. *Proceedings of the second IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 135 – 138, May 2004. [cited at p. 63, 64]

[39] Chao-Ju Hou and Kang G. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. *IEEE Transaction On Computers*, 46(12), DECEMBER 1997. [cited at p. 5]

[40] Mei-Chen Hsueh, T.K. Tsaib, and R.K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75 – 82, April 1997. [cited at p. 68]

[41] Farnam Jahanian. Fault-tolerance in distributed real-time systems. *IEEE Proceedings of the Third International Workshop on Real-Time Computing Systems Application*, page 178, 1996. [cited at p. 26]

[42] P. Jalote. *Fault Tolerance in distributed systems.* Prentice hall, 1994. [cited at p. 5, 24, 27]

[43] Shatha K. Jawad. Task scheduling in distributed systems using a mathematical approach. *Asian Journal of Information Technology*, 5(1):69–74, 2006. [cited at p. 59]

[44] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. *IEEE Real Time Systems Symposium*, pages 129–139, December 1991. [cited at p. 59]

[45] Philip H. Enslow Jr. Multiprocessor organization - a survey. *ACM Computer Survey*, 9(1):103–129, 1977. [cited at p. 3]

[46] J. L. T. Kim, K. H. Goldberg, and C. Subbaraman. The adaptable distributed recovery block scheme and a modular implementation model. *Proceedings of Fault-Tolerant Systems*, page 131 138, December 1997. [cited at p. 65]

[47] K. H. Kim. Slow advances in fault-tolerant real-time distributed computing. *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 106–108, October 2004. [cited at p. 3, 27, 63]

[48] K. H. (Kane) Kim. Middleware of real-time object based fault-tolerant distributed computing systems: Issues and some approaches. *Pacific Rim Int'l Symp. on Dependable Computing*, pages 3–8, December 2001. Keynote paper. [cited at p. 3, 43]

[49] K. H. (Kane) Kim. Toward integration of major design techniques for real-time fault-tolerant computer systems. *Transactions of the SDPS*, 6(1):83–101, March 2002. [cited at p. 3, 43]

[50] K.H. Kim. Designing fault tolerance capabilities into real-time distributed computer systems. *IEEE Proceedings of Workshop on the Future Trends of Distributed Computing Systems*, pages 318–328, September 1988. [cited at p. 21, 27, 64, 65]

[51] K.H. Kim. Importance of real-time distributed computing software building-blocks in realization of ubiquitous computing societies. *Proceedings of IEEE CS 7th Int'l Symp. on Autonomous Decentralized Systems*, April 2005. [cited at p. 3]

[52] Israel Koren. Fault tolerant computing. UNIVERSITY OF MASSACHUSETTS, 2005. [cited at p. 43]

[53] C.M. Krishna and Kang G. Shin. *Real Time Systems*. McGrawHill, 1997. [cited at p. 5, 17, 57, 63, 64, 66]

[54] C.M. Krishna and K.G. Shin. Performance measure for control computers. *IEEE Transctions on Automatic Control*, AC-32(6), June 1987. [cited at p. 17]

[55] J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82(1):25–40, January 1994. [cited at p. 3]

[56] J.C. Laprie. Dependable computing and fault tolerance : concepts and terminology. *IEEE 25th International Symposium on Fault-Tolerant Computing*, 3:27–30, June 1995. [cited at p. 21]

[57] Jean Claude Laprie, Jean Arlat, Christian Beounes, and Karama Kanoun. Definition and analysis of hardware and software fault tolerant architectures. *IEEE Computer*, 23(7):39 – 51, july 1990. [cited at p. 43]

[58] Ji Youn Lee, Soo-Yong Shin, Tai Hyun Park, and Byoung-Tak Zhang. Solving traveling salesman problems with dna molecules encoding numerical values. *BioSystems*, 78:39–47, 2004. [cited at p. 82]

[59] Peng Li and B. Ravindran. Efficiently tolerating failures in asynchronous real-time distributed systems. *7th IEEE International Symposium on High Assurance Systems Engineering, 2002. Proceedings*, pages 19 – 26, October 2002. [cited at p. 28]

[60] Antonios Litke, Dimitrios Skoutas, Konstantinos Tserpes, and Theodora Varvarigou. Efficient task replication and management for adaptive fault tolerance in mobile grid environments. *Future Generation Computer Systems*, 23:163–178, 2007. [cited at p. 5, 43]

[61] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM*, 20(1):46–61, 1973. [cited at p. 59]

[62] Jane W.S. Liu. *Real Time Systems*. Pearson Education, 8 edition, 2005. Indian Reprint. [cited at p. 51, 66]

[63] Jane W.S. Liu. *Real Time Systems*. Indian Reprint. Pearson Education, 8 edition, 2005. [cited at p. 59]

[64] V. Lo, K.J. Windisch, and B. Nitzberg. Noncontiguous processor allocation algorithms for mesh-connected multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):712–726, July 1997. [cited at p. 61]

[65] G. Manimaran and C.S.R. Murthy. A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1137 – 1152, November 1998. [cited at p. 43]

[66] P.M. Melliar-Smith and L.E. Moser. Progress in real-time fault tolerance. *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 109 – 111, 18-20 Oct 2004. [cited at p. 26]

[67] Daniel A. Menascé and Virgilio Almeida. Computing performance measures of computer systems with variable degree of multiprogramming. In *Int. CMG Conference*, pages 97–106, 1981. [cited at p. 11, 60]

[68] Bindu Mirle and Albert M. K. Cheng. Simulation of fault-tolerant scheduling on real-time multiprocessor systems using primary backup overloading. Technical Report UH-CS-06-04, Real-Time Systems Laboratory, Department of Computer Science, University of Houston, May 2006. [cited at p. 4, 43]

[69] G. Motel and J.C. Geffroy. Dependability computing:an overview. *Theoretical Computer Science*, pages 1115–1126, 2003. [cited at p. 21]

[70] Howard Jay Siegel Debra Hensgen Muthucumaru Maheswaran, Shoukat Ali and Richard F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Parallel and Distributed Computing*, 59:107–131, 1999. [cited at p. 57]

[71] M.D. Natale and J.A. Stankovic. Scheduling distributed real time task with minimum jitters. *IEEE Transaction On Computers*, 49(4):303–316, April 2000. [cited at p. 59]

[72] D. Nguyen and Dar-Biau Liu. Recovery blocks in real-time distributed systems. In *IEEE Proceedings of Annual Reliability and Maintainability Symposium*, pages 149 – 154, Jan. 1998. [cited at p. 36]

[73] S. Oh and G. MacEwen. Toward fault-tolerant adaptive real-time distributed systems, January 1992. [cited at p. 5]

[74] Taesoon Park and Heon Y. Yeom. Application controlled checkpointing coordination for fault-tolerant distributed computing systems. *Parallel Computing*, 26(4):467–482, 2000. [cited at p. 4, 28]

[75] A. Pataricza, I. Majzik nd W. Hohl, and J. Honig. Watchdog processors in parallel systems. *Microprocessing and Microprogramming*, pages 69–74, 1993. [cited at p. 34]

[76] Dar-Tzen Peng, Kang G. Shin, and Tarek F. Abdelzaher. Assignment and scheduling communicating periodic tasks in distributed real-time systems. *IEEE Transactions on Software Engineering*, 23(12):745–758, DECEMBER 1997. [cited at p. 5, 57]

[77] Juan R. Pimentel and Mario Salazar. Dependability of distributed control system fault tolerant units. *IEEE 28th Annual Conference of the Industrial Electronics Society*, 4:3164–3169, November 2002. [cited at p. 24]

[78] Aggeliki Prayati, Christos Koulamas, Stavros Koubias, and George Papadopoulos. A methodology for the development of distributed real-time control applications with focus on task allocation in heterogeneous systems. *IEEE Transactions on Industrial Electronics*, 51(6):1194–1027, December 2004. [cited at p. 57]

[79] S. Punnekkat and A. Burns. Analysis of checkpointing for schedulability of real-time systems. *Proceedings of IEEE Real-Time Systems Symposium*, pages 198–205, December 1997. [cited at p. 29]

[80] Li Qilin, Chen Yu, Zhou Mingtian, and Wu Yue. The research and implementation of a corba-based architecture for adaptive fault tolerance in distributed systems. *Proceedings of 5th International Conference on algorithms and Architectures for Parallel Processing*, pages 408 – 411, October 2002. [cited at p. 43]

[81] X. Qin and H. Jiang. Dynamic, reliability-driven scheduling of parallel real-time jobs in heterogeneous systems, 2001. [cited at p. 5]

[82] Xiao Qin, Zongfen Han, Hai Jin, Liping Pang, and Shengli Li. Real-time fault-tolerant scheduling in heterogeneous distributed systems. *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications*, I:421–427, June 2000. [cited at p. 4, 43, 64]

[83] Xiao Qin, Hong Jiang, and David R. Swanson. An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems. *International Conference on Parallel Processing*, pages 18–21, August 2002. [cited at p. 64]

[84] K. Ramamritham, P.F. Shiah, and J.A. Stankovic. Efficient scheduling algorithms for real time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1:184–194, 1990. [cited at p. 57]

[85] K. Ramamritham and J.A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of IEEE*, 82(1):55–67, January 1994. [cited at p. 58]

[86] K. Ramamritham, W. Zhao, and J.A. Stankovic. Distributed scheduling of tasks with deadline and resource requirements. *IEEE Transactions on Computers*, 38:1110–1123, 1989. [cited at p. 57]

[87] Krithi Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):412–420, April 1995. [cited at p. 5]

[88] S. Ramamurthy and M. Moir. Static-priority periodic scheduling of multiprocessors. *In Proc. of the 21st IEEE Real-Time Systems Symposium*, pages 69–78, December 2000. [cited at p. 5, 59]

[89] Ola Redell. Global scheduling in distributed real time computer systems. Department of Machine Design, Royal Institute of Technology, March 1998. [cited at p. 58]

[90] Ola Redell. Global scheduling in distributed real time computer systems: An automatic control perspective. Technical report, Royal Institute of Technology, Stockholms, 1998. [cited at p. 5]

[91] Christian Rehn. Dynamic mapping of cooperating tasks to nodes in a distributed system. *Future Generation Computer Systems*, 22:3545, 2006. [cited at p. 58]

[92] Yang ren Rau and Huey jenn Chiang. Dna computing on the minimum spanning tree problem. *International Symposium on Nanoelectronic Circuits and Giga-scale Systems (IS-NCGS 2004)*, pages 101–105, 2004. [cited at p. 81]

[93] G.G. Richard and M. Singhal. Using vector time to handle multiple failures in distributed systems. *IEEE Concurrency*, 5(2):50–59, Apr-Jun 1997. [cited at p. 28]

[94] R. Melhem S. Ghosh and D. Mosse. Fault-tolerant scheduling on a hard real-time multiprocessor system. *Proceedings., Eighth International Parallel Processing Symposium*, pages 775 – 782, April 1994. [cited at p. 4]

[95] Goutam Kumar Saha. Software based fault tolerant computing. *ACM Ubiquity*, 6(40):1, November 2005. [cited at p. 43]

[96] Beth A. Schroeder. On-line monitoring: A tutorial. *IEEE Computer*, 28(6):72–78, June 1995. [cited at p. 5]

[97] K.G. Shin and Y.C. Chang. Load sharing in distributed real time systems with state change broadcasts. *IEEE Transaction on Computers*, 38:1124–1142, 1989. [cited at p. 57]

[98] K.G. Shin and P. Ramanathan. Real time computing:a new discipline of computer science and engineering. *Proceedings of IEEE*, 82(1):6–24, January 1994. [cited at p. 5, 52]

[99] S.Y. Shin, B.T. Zhang, and S.S. Jun. Solving traveling salesman problems using molecular programming. *Proceedings of the Congress on Evolutionary Computation*, 2:994–1000, 1999. [cited at p. 82]

[100] Sang H. Son, Fengjie Zhang, and Ji-Hoon Kang. Replication control for fault-tolerance in distributed real-time database systems. *12th International Conference on Distributed Computing Systems*, pages 144–151, June 1992. [cited at p. 43, 65]

[101] J. A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10):10–19, 1988. [cited at p. 47]

[102] J. A. Stankovic. Distributed real time computing:the next generation. Technical report, University of Massachusetts, January 1992. [cited at p. 5, 52]

[103] Paul Stelling, Cheryl DeMatteis, Ian T. Foster, Carl Kesselman, Craig A. Lee, and Gregor von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128, 1999. [cited at p. 27, 43]

[104] Dong Tang and Ravishankar K. Iyer. Dedendabilitv measurement and modeling of a multicomputer system. *IEEE Transaction on Computers*, 42(1), January 1993. [cited at p. 21]

[105] C. Tanzer, S. Poledna, E. Dilger, and T. Fhrer. A fault-tolerance layer for distributed fault-tolerant hard real-time systems. [cited at p. 4, 43]

[106] Nguyen Duc Thai. Real-time scheduling in distributed systems. *IEEE Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, 2002. [cited at p. 61]

[107] Van Tilborg and A.M. Koob. *Foundations of Real Time Computing:Scheduling and Resource Management*. Kluwer Academic Publishers. [cited at p. 66]

[108] Sebastien Tixeuil, William Hoarau1, and Luis Silva. An overview of existing tools for fault-injection and dependability benchmarking in grids. CoreGRID Technical Report TR-0041, Institute on System Architecture, CoreGRID - Network of Excellence, October 2006. [cited at p. 68]

[109] T. Tsuchiya, Y. Kakuda, and T. Kikuno. Fault-tolerant scheduling algorithm for distributed real-time systems. *Proceedings of the Third Workshop on Parallel and Distributed Real-Time Systems*, pages 99 –103, April 1995. [cited at p. 65]

[110] Andy Tyrrell. Biologically inspired fault-tolerant computer systems. Technical Report YO10 5DD, Bio-Inspired Architectures Laboratory, Department of Electronics, University of York. [cited at p. 43]

[111] Degeng Wang and Michael Gribskov. Examining the architecture of cellular computing through a comparative study with a computer. *Journal of The Royal Society Interface*, 2(3), June 2005. [cited at p. 82]

[112] Wong and Dillon. A fault tolerant model to attain reliability and high performance for distributed computing on the internet. *Computer Communications*, 23(18):1747–1762, December 2000. [cited at p. 4]

[113] Jen yao Chung, Jane W. S. Liu, and Kwei-Jay Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transaction On Computer*, 39(9), September 1990. [cited at p. 59]

[114] Yongbing Zhang, K. Hakozaki, H. Kameda, and K. Shimizu. A performance comparison of adaptive and static load balancing in heterogeneous distributed systems. *Proceedings of the 28th Annual Simulation Symposium*, pages 332 – 340, April 1995. [cited at p. 64]

[115] Qin Zheng and Kang G. Shin. Fault-tolerant real-time communication in distributed computing systems. *IEEE Transactions On Parallel And Distributed Systems*, 9(5), MAY 1998. [cited at p. 43]

[116] Qin Zheng, B. Veeravalli, and Chen-Khong Tham. Fault-tolerant scheduling of independent tasks in computational grid. *Proceedings of 10th IEEE Singapore International Conference on Communication systems*, pages 1–5, October 2006. [cited at p. 64]

[117] Jeffrey J. P. Tsai Zhiying Wang and Chunyuan Zhang. Fault-toleration by duplication and debugging for distribution real-tine systems. *Tamkang Journal of Science and Engineering*, 3(3):187–194, 2000. [cited at p. 4]

Thesis contribution

1. Bibhudatta Sahoo and Aser Avinash Ekka, "Performance analysis Performance Analysis Of Concurrent Tasks Scheduling Schemes In A Heterogeneous Distributed Computing System", *Proceedings of National Conference on CS&IT*, KIET Ghaziabad, pp. 11-21, November 2006. URI: http://hdl.handle.net/2080/352

2. Bibhudatta Sahoo and Aser Avinash Ekka, "A Novel Fault Tolerant Scheduling Technique In Real-Time Heterogeneous Distributed Systems Using Distributed Recovery Block", *Proceedings of National Conference On High Performance Computing*, Government College Of Engineering, Tirunelveli, pp. 215-219, April 2007. URI: http://hdl.handle.net/2080/417

3. Bibhudatta Sahoo and Aser Avinash Ekka, "Distributed Recovery Block based Fault-Tolerant Dynamic Scheduling scheme for Real Time Heterogeneous Distributed System", *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA'07, Las Vegas, USA, June 2007. URI: http://hdl.handle.net/2080/441

4. Bibhudatta Sahoo and Aser Avinash Ekka, "Backward Fault Recovery in Real Time Distributed System of periodic tasks with Timing and Precedence Constraints", *Proceedings of International Conference on Emerging Trends in High Performance Architecture Algorithms & Computing*, pp.: 124-130, Chennai, India, July 2007. URI: http://hdl.handle.net/2080/447

5. Bibhudatta Sahoo and Aser Avinash Ekka, "Design and Performance of Fault-Tolerant Dynamic Scheduling Schemes for in Real Time Heterogeneous Distributed System", Accepted in *International Conference on Information Processing*, Bangalore, India, August 2007.

6. Bibhudatta Sahoo and Aser Avinash Ekka, "Fault Tolerant Real Time Dynamic Scheduling Algorithm For Heterogeneous Distributed System", Communicated to *IEEE International conference on Parallel and Distributed Systems*, Taiwan, CHINA, December 2007.